DYNAMIC SHARED MEMORY ARCHITECTURE, SYSTEMS, AND OPTIMIZATIONS FOR HIGH PERFORMANCE AND SECURE VIRTUALIZED CLOUD

A Thesis
Presented to
The Academic Faculty

by

Qi Zhang

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science

Georgia Institute of Technology
May 2017

DYNAMIC SHARED MEMORY ARCHITECTURE, SYSTEMS, AND OPTIMIZATIONS FOR HIGH PERFORMANCE AND SECURE VIRTUALIZED CLOUD

To Mom and Dad

ACKNOWLEDGEMENTS

I gratefully acknowledge the invaluable support I have received from my advisor, Professor Ling Liu, over the course of my doctoral study. Her endless enthusiasm for research and unsurpassed energy have inspired me to work on several interesting and practical research problems. In addition to academic advising, she has shared her life lessons with me and always been available whenever I needed help. I have been extremely fortunate to have her "in my corner" as my doctoral advisor, and I hope to pass the lessons I learned from her on to my students in the future.

I would also like to express my thanks to my doctoral dissertation committee members: Professors Calton Pu, Mustaque Ahamad, Douglas Blough, and Jay Lofstead. Their insightful comments and suggestions on my research have not only greatly contributed to my thesis but also helped broaden my horizons for my future research. I have also been fortunate to spend one summer at IBM Almaden and three summers at IBM Research T.J. Watson as a summer research intern and experience real-world research and engineering challenges. I express sincere thanks to my IBM mentors and collaborators including Donna Dillenberger, Gong Su, Arun Iyengar, Ashish Kundu, Aameek Singh, and Pramod Mandagere.

I would like to thank every member of the DiSL Research Group, Databases Laboratory, and Systems Laboratory at Georgia Tech for their collaboration and companionship. It was a great pleasure to work in such a dynamic research environment. I convey special thanks to Kisung Lee, Wenqi Cao, Semih Sahin, Emre Yigitoglu, Lei Yu, and Yang Zhou for countless research discussions and friendship. I have also been exceptionally fortunate to meet many wonderful friends in Atlanta.

Finally, and most importantly, I would like to thank my wife Jingxuan Liu. Her support and encouragement were undeniably the bedrock upon which the past five years of my life have been built. Her tolerance of my occasional vulgar moods is a testament in itself of her unyielding devotion and love. I thank my parents, Zhongyuan Zhang and Guilan Gong, for their faith in me and allowing me to be as ambitious as I wanted. It was under their watchful eye that I gained so much drive and an ability to tackle challenges head on. Also, I thank Jingxuan's parents, Xiaoping Liu and Qixia Wu. They also provided me with unending encouragement and support.

Contents

List of Tables

List of Figures

SUMMARY

Dynamic memory consolidation is an important enabler for high performance virtual machine (VM) execution in virtualized Cloud. Efficient just-in-time memory balancing requires three core capabilities: (i) Detecting memory pressure across VMs hosted on a physical machine; (ii) Allocation of memory to respective VMs; (iii) Enabling fast recovery upon making newly allocated memory available at the high pressure VMs. Although the Balloon driver technology facilitates the second task, it remains difficult to accurately predict the VM memory demands at affordable overhead, especially under unpredictable and changing workloads. Furthermore, no prior study analyzed the effect of slow response of VM execution to the newly available memory due to paging based application recovery.

In this dissertation research, I have made four original contributions to dynamic shared memory management in terms of architecture, systems and optimizations for improving VM execution performance and security. First, we designed and developed *MemPipe*, a shared memory inter-VM communication channel for fast inter-VM network I/O. *MemPipe* increases the shared memory utilization by adaptively adjusting the shared memory size according to workloads demands. It also reduces the inter-VM network communication overhead by directly copying the packets from the sender VM's user space to the shared memory area. Second, we developed *iBalloon*, a light-weight and transparent prediction based facility to enable automated or semi-automated ballooning with more customizable, accurate, and efficient memory balancing policies among VMs. Third, we developed *MemFlex*, a novel shared memory swapping facility that can effectively utilizes host idle memory by a hybrid memory swap-out model and a fast swap-in optimization. Fourth, we introduced *SecureStack*, which is a kernel backed tool to prevent the sensitive data on the function stack from being illegally accessed by the untrusted functions. *SecureStack* introduces three procedures to protect, restore, and clear the stack in a reliable and low cost manner. It is highly transparent to the users and does not bring any new vulnerability to the existing system.

The above research developments are packaged into *MemLego*, a new memory management framework for memory-centric computing in the big data era.

Chapter I

INTRODUCTION

With the increasing demands for fast big data processing, memory is becoming one of the most critical resources to improve the performance of the various applications in the Cloud. Large size memory machines and platforms are not uncommon nowadays, and many applications can benefit from using large memory to avoid expensive disk I/O operations. For example, Amazon is providing a single node with 244GB memory and Oracle has built a machine with 32TB memory. At the same time, in memory databases such as Redis and memory centric big data platforms such as Spark have been widely deployed in the Cloud.

However, how to efficiently and securely use such large memory is still an open issue. Maximizing the memory utilization while maintaining QoS for individual applications and guaranteeing the in-memory data security is still challenging to the existing Cloud providers. On one hand, recent traces from production data centers have revealed big gaps between resource allocation and utilization. Taking memory for example, total allocation account for more than 90% of a cluster's memory capacity whereas the overall usage is less than 50%. Idle memory is detrimental to the Cloud in terms of both cost and energy. Despite the fact that power consumption of idle CPU can be decreased by DVFS, idle memory occupies upto 40% energy consumption of a whole machine [12]. Current VM and application consolidation algorithms usually lead to poor performance because of the resource contention, especially on memory subsystem. Large amount of Performance variation for key data center services, such as web search, content analyzer, cloud storage, message communication, has been observed due to contention at all levels of memory hierarchy. Other than performance, security is another big concern for both Cloud providers and consumers. While large amount of untrusted APIs are used in the software development, in-memory sensitive data leakage is becoming huge threat. For example, the Heartbleed [33] is a known security bug in the OpenSSL cryptograph library. The compromised library can read more data than it should be allowed.

To tackle these challenges, this dissertation research is focused on the development of architectures and optimization techniques for a high performance and secure virtualization environment, especially in the era of Cloud computing.

1.1 Technical Challenges

We describe the technical challenges to build a high performance and secure virtualized Cloud from the perspective of memory resource management in more details as follows.

1.1.1 Memory Utilization

Dynamic memory consolidation is an important and attractive functionality to improve the memory utilization while preserve high performance applications in a virtualization platform. Ballooning is a dynamic memory balancing mechanism for non-intrusive sharing of memory between host and its guest VMs through a balloon driver with inflation and deflation operations. However, it is hard to make decisions on when to start ballooning and how much memory ballooning is sufficient. The state of art proposals typically resort to estimating the working set size of each VM at run time. Based on its estimated memory demands, additional memory will be dynamically added to or removed from the VM [42, 73, 75, 124]. However, accurate estimation of VM working set size is difficult under changing workloads [39]. Therefore, dynamic memory balancer may not discover in time that the VM is under memory pressure, or may not balloon adding memory fast enough. Also, by virtualization design, when a virtualized host boots, it treats each of its hosted VMs as a process and allocates it a fixed amount of memory. Each guest VM is managed by a guest OS, independently (and unaware of the presence) of the host OS. Thus, even when the host has sufficient free memory, the guest VMs under memory pressure are unaware. Therefore, any delay in dynamic memory balancing can cause the VM not be able to utilize the host idle memory in a timely fashion. Therefore, we argue that, in a virtualized environment, the memory resource can be used more effectively to further improve various aspects of VM performance, such as inter VM communication and VM memory swap.

1.1.2 Memory Balancing

Lots of existing researches are focused on exploring how to dynamically balance the memory allocation to meet a VM's changing demand. Drawing from the lessons and experiences of previous work, we believe that there are two common issues need to be addressed: (1) monitoring the VM resource demand at a low cost to decide when and where to move memory among the VMs; (2) moving memory among the VMs with minimal impact on the performance of the VMs. Existing researches [119] [100] [133] have proposed many methods to predict the VM memory utilization. However, an accurate prediction of VM memory working set size is still a difficult problem, especially under changing conditions [39]. Because of the fact that hypervisor lacks the knowledge of VM memory access pattern, virtualization environment makes this prediction even harder. The second issue has been partially addressed by the introduction of memory balloon driver [120], which allows memory to be moved among the co-located VMs and the host machine. However, balloon driver cannot work by itself. In other words, system administrators have to be involved to periodically check the memory utilization of each VM and make the decision of how to balance the memory around. There are actually some efforts to make it work automatically [2], but the system is still in its initial stage and there lacks extensive experiments to evaluate its performance. Although some researchers propose ideas to rebalance memory among VMs by using balloon driver [113] [133], they also require guest kernel modification and the overhead incurred by memory access interception in these approaches can be very high.

1.1.3 Memory Security

Using the third-party libraries is a common approach to facilitate the software development. However, third-party libraries are also becoming one of the most *insecure* components of an application [28], since it can be utilized to hack the sensitive data on the application's stack. Existing research efforts on protecting stack data leakage have centered on two most well-known vulnerabilities: the uninitialized read problem and the stack overflow attack. However, we argue that using an untrusted function (e.g. third party functions) can be a new vulnerability of stack data leakage. Since a function has access to the stack of the whole process it belongs to, it can look beyond its own stack frames. Sensitive data can be leaked when an untrusted function is invoked while stack frames of the functions holding sensitive data are already present in the process stack. Specifically, since RSP and RBP are two general registers on the x86_64 platform that can be modified from user level, an attacker can simply obtain the stack boundaries from the RSP and RBP registers and scan the stack memory accordingly. In this case, the attacker can readily bypass the existing state of art approaches to gain access to the sensitive data on the stack, in addition to reading from the uninitialized variables and compromising return addresses to launch the stack overflow attack. Programmers can suffer from such sensitive stack data leakage attacks when they are using untrusted functions in third-party libraries. Since these libraries are not fully developed by the users, and are under the risk of being compromised. When a user invokes a compromised library call within a function or after a function, which contains sensitive data on its stack, the sensitive data can be disclosed.

1.2 Dissertation Scope and Contributions

In this dissertation, we propose solutions to enable more efficient and secure memory resource utilization. We achieve our goal from the following aspects:

First, we present *MemPipe*, a dynamic shared memory pipe framework for efficient data transfer between co-located VMs. MemPipe is novel in the following aspects: first, *MemPipe* provides high performance and elastic memory management. Instead of statically and equally allocating a shared memory pipe to each pair of co-located communicating VMs, *MemPipe* slices the shared memory into small chunks and allocates the chunks proportionally to each pair of VMs based on their runtime demands. We show that this dynamic proportional memory allocation mechanism can significantly enhance the utility of shared memory channels while improving the co-located inter-VM network communication performance. Second, *MemPipe* introduces two optimization techniques - *time-window based streaming partitions* and *socket buffer redirection* to enhance the performance of inter-VM communication for streaming networking workloads and eliminates the network packet data copy from sender VM's user space to its VM kernel.

Second, we design *MemFlex*, a flexible shared memory swapper with three original contributions: (1) By redirecting the VM memory swapping to the host-guest shared memory swap partition, *MemFlex* avoids the high overhead of disk I/O for guest swapping as well as guest-host context switching, and enables the guest VM to respond fast to the newly ballooned memory and to quickly recover from severe performance degradation under peak memory demands. (2) To handle the situation of limited shared memory swap area due to insufficient available memory at the host, *MemFlex* provides a hybrid memory swapping model, which treats shared memory swap partition as the small primary swap area and the disk swap partition(s) as the secondary swap area. This model enables fast shared memory based VM swapping whenever it is possible and a smooth transition to the conventional guest OS swapping on demand. (3) To address the problem of slow recovery of memory intensive workloads even after sufficient additional memory has been successfully allocated via ballooning, we provide a fast swap-in optimization to proactively swap-in the pages resident in the shared memory swap area, reducing the high cost of frequent paging based swap-in.

Third, we develop *iBalloon*, which is a low cost VM memory balancer with high accuracy and transparency. No modification is required for VMs or the hypervisor to deploy *iBalloon*, which makes it more acceptable in real cloud environment. The goal of *iBalloon* is to to keep a balanced memory utilization among VMs running on the same host while avoiding any VM from being deprived of free memory, with low cost and high accuracy and transparency. *iBalloon* consists of a Per-VM Monitor and a Balancer. Both the Per-VM Monitor and the Balancer are user level daemons. The Per-VM Monitor, which runs in the user space of each VM, is responsible for collecting information about the memory utilization of this VM. The Balancer, which consists of three parts: VM Classifier, Memory Balancer, and Balloon Executor, executes in the user space of the host. By using the *Exponentially Weighted Moving Average (EWMA)* model, the Balancer reads the information collected by the Per-VM Monitor, predicts each VM's future memory utilization, and makes decisions about how to rebalance the memory among the VMs. The Balancer then contacts the balloon driver to actually move memory among the VMs. Communications between the Per-VM

Monitor and the Balancer are via an in memory bitmap and shared files, which are located on host and exported to the VMs by the Network File System (NFS).

Fourth, we introduce *StackVault*-a kernel-backed system-level facility to eliminate sensitive stack data leakage. This work is motivated by real applications scenarios that software development team at many companies face when writing large-scale applications with sensitive data [20]. Examples of such data include protected health information(PHI) and financial records. Concretely, *StackVault* enforces three types of stack protection operations to protect the sensitive stack data by preventing an untrusted function from illegally accessing the stack of another function in the same process. Through placement and enforcement of such operations, *StackVault* moves the sensitive stack data into an OS kernel buffer prior to the execution of an untrusted function, so that there is no way for such data to be touched by any untrusted function. Such protection also ensures that all data required for execution of the untrusted function is kept on the stack. The stack data is restored immediately after the untrusted function returns, and the stack is cleared for every sensitive function upon its return, in order to eliminate any leakage of stack data after its completion. We assume that the OS kernel is the trusted computing base, and any buffer in the kernel cannot be accessed by the attacker in the user level.

1.3 Dissertation Organization

This dissertation consists of several chapters and each chapter addresses one or more of the problems described above. In each chapter, we introduce the background of the problem being addressed, describe related work, and present our solution techniques followed by experimental evaluation. This dissertation is organized as follows.

In Chapter 2, we present the design and implementation of *MemPipe*, a dynamic shared memory management system for high performance network I/O among virtual machines (VMs) located on the same host. *MemPipe* employs an inter-VM shared memory pipe to enable high throughput data delivery for both TCP and UDP workloads among co-located VMs, manages its shared memory pipes through a demand driven and proportional memory allocation mechanism, and employs a number of optimization techniques such as *time-window based streaming partitions* and *socket buffer redirection* to further improve the performance of co-located inter-VM communication.

In Chapter 3, propose *iBalloon*, a lighted-weighted, high accurate and transparent VM memory balancing service. *iBalloon* consists of two major components: the Per-VM Monitor and a global Balancer. *iBalloon* predicts the VM memory utilization based on Exponentionally Weighted Moving Average (EWMA) model and dynamically adjust the VM memory accordingly.

In Chapter 4, we introduce the design of *MemFlex*, a highly efficient shared memory swapper. *MemFlex* can effectively utilize host idle memory by redirecting the VM swapping traffic to the host-guest shared memory swap area. The hybrid memory swapping model in *MemFlex* promotes to use the fast shared memory swap partition as the primary swap area whenever possible, and smoothly transits to the conventional disk-based VM swapping scheme on demand. Also, *MemFlex* proactive swap-in optimization offers just-in-time performance recovery by replacing costly page faults with an efficient swap-in implementation.

In Chapter 5, we describe the design and implementation of *MemLego*, a shared memory based memory optimization framework for managing and improving memory efficiency in virtualized environment. *MemLego* offers on-demand VM memory allocation and deallocation in the presence of changing workloads. maintains a shared memory region across multiple VMs, and enables those VMs under memory pressure to obtain additional memory on demand.

In Chapter 6, we develop *StackVault*, which is a highly reliable and transparent tool to protect the sensitive data on the function stack. *StackVault* developed a novel and unforgeable function identity to prevent an untrusted function to steal data from a protected stack. It also has a three-phase framework in order to secure sensitive data on the stack: (1) capturing application-specific sensitive functions and untrusted functions through easy-to-use configurations, (2) transparent placement of stack protection operations through system-supplied secure procedures to protect, restore, and clear the stack, and (3) automated enforcement of stack protection through spatial and temporal access monitoring and control over both sensitive stack data and untrusted functions.

In Chapter 7, we summarize the main contributions of this dissertation and discuss our future research directions.

Chapter II

WORKLOAD ADAPTIVE SHARED MEMORY MANAGEMENT FOR HIGH PERFORMANCE NETWORK I/O IN VIRTUALIZED CLOUD

This chapter presents the design and implementation of MemPipe, a dynamic shared memory management system for high performance network I/O among virtual machines (VMs) located on the same host. MemPipe delivers efficient inter-VM communication with three unique features. First, MemPipe employs an inter-VM shared memory pipe to enable high throughput data delivery for both TCP and UDP workloads among co-located VMs. Second, instead of static allocation of shared memories, MemPipe manages its shared memory pipes through a demand driven and proportional memory allocation mechanism, which can dynamically enlarge or shrink the shared memory pipes based on the demand of the workloads in each VM. Third but not the least, MemPipe employs a number of optimizations, such as time-window based streaming partitions and socket buffer redirection, to further optimize its performance. Extensive experiments show that MemPipe improves the throughput of conventional (native) inter VM communication by up to 45 times, reduces the latency by up to 62%, and achieves up to 91% shared memory utilization.

2.1 Introduction

Network I/O is known to be the dominating workloads in virtualized clouds. Achieving high performance inter-VM communications is a key challenge for many Cloud services, systems and applications. On one hand, virtualization by design introduces host-neutral abstraction. This enables applications transparently communicate across VM boundary using standard TCP/IP sockets and increases the flexibility of VM management. On the other hand, these gains have been thwarted by the network I/O performance in virtualized clouds: for the data residing on the same host, the network communication overhead between co-located VMs can be as high as the communication cost between VMs located on different hosts [121, 131]. This is because all network packets transmitted from sender VM to receiver VM have to travel through the boundaries of VMs via hypervisor.

Research efforts in systems virtualization area have been engaged in reducing the cost of data copies across the VM boundaries using shared memory based solutions [131, 108]. Instead of going through traditional network stack, communicating through shared memory shortens the communication path, avoids the barrier cost of hypervisor, and improves the data transmission efficiency. Although existing proposals [121, 77, 81, 44, 106, 132, 95] differ from one another in terms of concrete design and implementation decisions, most of these efforts suffer from some of the following problems: poor scalability in terms of shared memory management for different types of workloads and dynamic VM deployment[97, 89], and multiple copies of network packet between VM kernel buffer and the shared memory.

Recent research on network I/O virtualization has centered on improving the inter-VM network I/O performance by software defined network (SDN) and network function virtualization (NFV). Representative technology includes the single root I/O virtualization (SR-IOV) [18] for making PCI devices interoperable, and Intel Data Plane Development Kit (DPDK) [10, 5] for fast packet processing using multicore systems. SR-IOV capable devices allow multiple VMs to independently share a single I/O device and can move data from/to the device by bypassing the virtual machine monitor(VMM). Although SR-IOV improves the communication between VM and its physical device, it cannot remove the overhead of co-located inter-VM communication. This is because with SR-IOV, packets still need to travel through the network stack of the sender VM, to be sent from the VM to the SR-IOV device, and then sent to the receiver VM. This long path can still lead to unnecessarily high cost for co-located inter-VM communication, especially for larger sizes messages. Alternatively, Intel DPDK is a set of libraries and drivers, which utilizes huge pages in guest VMs and multicore processing to provide applications direct access to the packets on NIC. DPDK is restricted and on its own cannot yet support flexible and fast network functions [5]. NetVM [78] develops a shared memory based approach for KVM using DPDK to show that the virtualized edge servers can provide fast packet delivery to VMs without passing through the hypervisor. However, NetVM is limited to run on a DPDK enabled multicore platform and no open source is made available to date. Another

VM communication optimization approach is Virtio [110], which is an abstraction layer over devices, including NIC, in a paravirtualized hypervisor. For a specific device, Virtio implements a front-end driver in the guest VM and a back-end driver in the host. A common set of interfaces are also defined to enable the front-end driver and back-end driver to communicate with each other. It is designed for efficient communication between the VMs and the host hardware, but not for inter VM communication. Packets transferred between co-located VMs still have to go through the host.

In this chapter, we present MemPipe, a dynamic shared memory pipe framework for efficient data transfer between co-located VMs. MemPipe is novel in the following aspects: *First*, MemPipe provides high performance and elastic memory management. Instead of statically and equally allocating a shared memory pipe to each pair of co-located communicating VMs, MemPipe slices the shared memory into small chunks and allocates the chunks proportionally to each pair of VMs based on their runtime demands. We show that this dynamic proportional memory allocation mechanism can significantly enhance the utility of shared memory channels while improving the co-located inter-VM network communication performance. *Second*, MemPipe introduces two optimization techniques - *time-window based streaming partitions* and *socket buffer redirection* to enhance the performance of inter-VM communication for streaming networking workloads and eliminates the network packet data copy from sender VM's user space to its VM kernel. We implement MemPipe on KVM, have detailed challenges we have solved and tradeoffs we have made, and have released MemPipe as an open source system.

The rest of this chapter is organized as follows: Section 2.2 introduces the shared memory approach and discusses the inherent problems in existing shared memory implementations. We describe the design choices of MemPipe in Section 2.3 and the implementation details of MemPipe in Section 2.4. We evaluate the performance of MemPipe in Section 2.5, discuss open issues in Section 2.6, review the related work in Section 2.7, and conclude the chapter in Section 2.8.

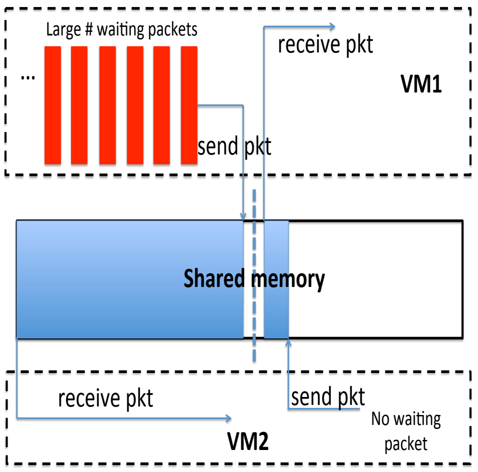


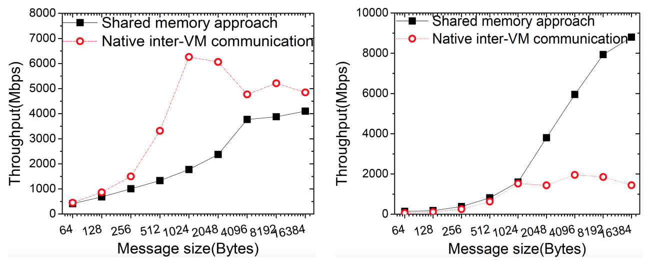
Figure 1: Co-located VM communication via static shared mem.

2.2 Background

How does shared memory work? Shared memory mechanism is first introduced to pass data between programs that are running on the same operating system to avoid redundant copy. Inspired by this idea, shared memory approach has been developed to accelerate the performance of co-located inter VM communication [121, 77, 81, 44, 106, 132, 95]. The general idea is to transmit data from a sender VM to a co-located receiver VM by using the shared memory channel and bypass the hypervisor. Concretely, instead of delivering network packets via the host operating system or hypervisor, the sender VM will intercept and analyze the packets, and then redirect them to a pre-allocated memory region that is shared with the receiver VM. After putting the packets into the shared memory, a notification is sent to the receiver VM for the packet to be picked up. Using shared memory based inter-VM communication, the sender VM's packets can be made visible immediately to the receiver without going through the hypervisor.

Problems of existing implementations. Although shared memory mechanism holds the potential of high communication efficiency for co-located VMs, existing implementations of shared memory based approach fail to deliver its full potential due to a number of limitations.

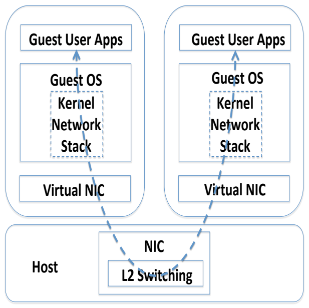
First, most of existing shared memory based approaches use the static and equal allocation method for shared memory management. By allocating the same amount of shared memory for each pair of communicating VMs regardless the workloads' demands, it can lead to inefficient utilization of shared memory resource. Furthermore, static shared memory management cannot adapt to the changing demand for shared memory and leads to poor availability for skewed workloads. Figure 1 depicts a scenario when two co-located VMs are communicating via equally and statically divided shared memory: Left portion of the shared memory is for VM1 as the sender and VM2 as the receiver, while right portion is for VM2 as the sender and VM1 as the receiver. Given that VM1 is sending much more data packets to VM2 whereas VM2 is sending only short acknowledgement messages to VM1, the static shared memory management may result in several problems: (1) a long waiting queue in VM1 to deliver the packets; and (2) inefficient utilization of shared memory resource. When the sending rate of the packets on VM1 is too fast, the shared memory pipe may become too full to take more packets, causing packets drop, even though the overall shared memory is under utilized due to the low shared memory utilization in VM2. However, simply allocating different amount of shared memory to different VMs prior to runtime is still problematic and not scalable, since the workloads in each VM may vary from time to time. We conjecture that workload adaptive shared memory allocation and management are mandatory for providing highly available and highly scalable inter-VM communication channels.



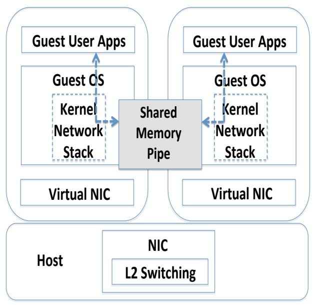
(a) TCP streaming workload (b) UDP streaming workload

Figure 2: Comparison of Inter-VM communication using shared memory v.s. using the conventional network stack(referred to as Native Inter-VM communication).

Another critical issue that has not been investigated is whether the shared memory



(a) Native inter-VM



(b) MemPipe

Figure 3: Packets delivery path

based approach has consistent performance under different types of workloads. For example, network I/O workloads in virtualized Clouds can be broadly categorized into four types: TCP streaming or TCP transaction workloads and UDP streaming or UDP transaction workloads. Some of the existing implementations of shared memory mechanism demonstrate better performance than conventional (native) inter-VM communication only for UDP workloads. Figure 2 compares the performance of inter VM communication using the conventional (native) inter-VM communication approach (see the blue line with solid squares) with using a shared memory approach (see the red line with empty circles). Both TCP and UDP streaming workloads generated by Netperf [14] are used in this set of experiments. We vary the message sizes from 64B to 16KB in the X-axis and measure the inter VM communication throughput of TCP and UDP workloads, shown in Y-axis of Figure 2(a) and (b) respectively. We make two observations. First, Figure 2(b) shows that for UDP_STREAM workloads, the shared memory based approach offers up to 6 times higher performance than native co-located inter VM communication. The performance gain increases as the message size increases. Second, we observe the opposite results: the performance of TCP_STREAM workloads by shared memory approach is consistently worse than that of native (conventional) inter-VM communication, irrespective of the message sizes. The main cause for this performance discrepancy under TCP workloads is due to the lack of certain TCP packet level optimizations in the existing shared memory systems (see more detailed discussion in Section 5.3). Note that Xenloop [121] is chosen as the shared memory approach for this set of experiments among several other shared memory systems, such as MMNet [106], XWAY [81], IVC [77], because it is widely considered as one of the most representative shared memory systems in terms of best practice [108]: (i) XenLoop is transparent to the user level program, the guest OS kernel and the virtual machine monitor, (ii) XenLoop supports both TCP and UDP workloads unlike XWAY, which supports only TCP workloads, and (iii) XenLoop is opensourced, whereas many other similar systems are not.

Although all existing shared memory approaches dedicate to utilizing the shared memory mechanism to remove the cost of data copy between VM and hypervisor, few has explored the opportunities to further remove the cost of data copying between guest VM kernel and shared memory. Concretely, even with zero copy between VM and hypervisor, for a sender VM to send a packet to a co-located receiver VM via shared memory, the packet still needs to be copied 4 times: (1) from sender VM's user space to its kernel socket buffer, (2) from sender VM's kernel buffer to its shared memory, (3) from shared memory to receiver VM's kernel socket buffer, and (4) from receiver's kernel buffer to the application buffer. In MemPipe, we further reduce the cost of data copying by allowing the sender VM kernel to directly copy the packets from the user level buffer to the shared memory.

2.3 MemPipe System Design

To better illustrate the system design of MemPipe, we provide Figure 3 to compare the existing network I/O processing in conventional virtualization platform (i.e., native inter-VM) with MemPipe in terms of packet delivery path between sender VM and receiver VM.

In native inter-VM communication, packets are sent from a sender VM through guest OS kernel and hypervisor to the physical NIC. Three copies are performed at the sender VM: user space to kernel space in the sender VM, sender VM kernel to hypervisor (host OS), and then onto the NIC via hypervisor (host OS kernel). Upon arrival at the NIC, a virtual switch performs L2 switching to determine the receiver VM for each packet and notifies the appropriate virtual NIC. At the receiver VM, another two copies are performed: the memory page containing the packet is either copied or granted to the kernel space of its Guest OS, and then the data is copied from guest OS kernel to the user space of the guest OS, from which the application of interest can access the data. Such long data packet transmission path involves significant overhead, leading to the poor latency and low throughput.

The inter-VM communication path in MemPipe is shown in Figure 3(b). By using MemPipe, data packets can be sent from the sender VM to the receiver VM directly with zero-copy at the hypervisor level, enabling applications at the sender or receiver VM to write or read the data packets with much shorter network I/O latency. Mempipe also provides the *socket buffer redirection* capability for applications to opt-in and opt-out by choice, which further reduces the data copy between co-located VMs.

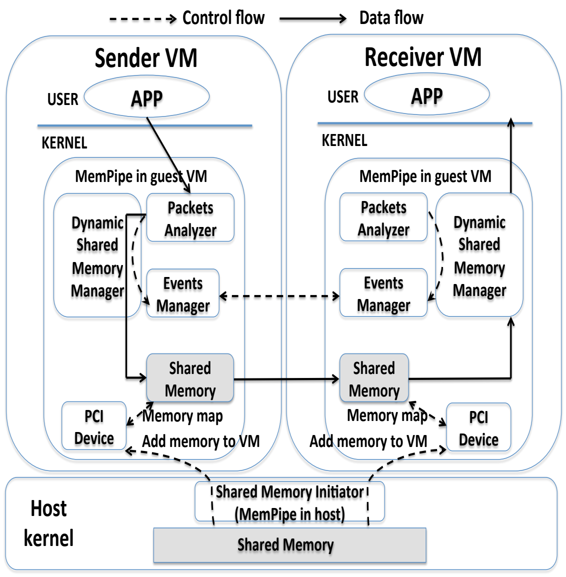


Figure 4: MemPipe system overview

2.3.1 Shared Memory Allocation

Shared memory pipes can be allocated globally by the host OS (hypervisor) or from each VM. In MemPipe, the shared memory for co-located inter VM communication is globally pre-allocated by the host OS (hypervisor). We choose this global shared memory allocation mechanism instead of allocating shared memory within each VM for a number of reasons. First, the presence of MemPipe will not increase the memory pressure within a single VM. Second, allocating shared memory in the host (hypervisor) increases the reliability of

MemPipe in the presence of guest domain failure. For example, if the shared memory is allocated within a sender VM, upon a sudden crash of the sender VM, if the co-located receiver VM tries to access the shared memory region of the sender VM, which no longer exists, it will lead to kernel panic.

In addition, MemPipe promotes dynamic shared memory allocation. Instead of equally dividing the shared memory among co-located VMs, MemPipe divides the whole piece of shared memory into small *chunks* of fixed size. Initially, each memory pipe in a VM is assigned with one chunk. Based on the workload demand, more chunks can be assigned to or revoked from memory pipes dynamically. This proportional management of global shared memory using small chunks allows the growing and shrinking of shared memory pipes on demand for high performance and high availability.

2.3.2 The Lifecycle of Packets in MemPipe

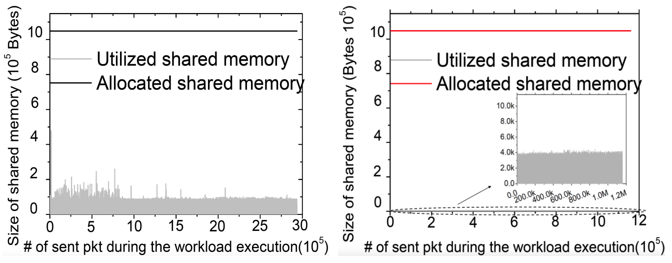
Figure 4 illustrates the delivery path of a network packet from a sender VM to a receiver VM by MemPipe. The solid lines represents the data flow while the dotted line shows the control flow. The MemPipe system consists of three core functional components: packet analyzer, event manager and dynamic shared memory manager.

Packet analyzer intercepts the outgoing packets of each sender VM and checks whether a packet is heading to a co-located VM. If not, the packet will be sent through the default kernel path, otherwise MemPipe delivery path will be used to deliver the packet. Events manager delivers interrupt notifications between the sender VM and the receiver VM. When a packet is intercepted and identified by the MemPipe packet analyzer as to be sent to a co-located VM, MemPipe notifies the VM kernel to allocate the socket buffer for this packet from shared memory instead of the default kernel buffer, and the data packet will be directly copied from the user level application buffer to the shared memory. The events manager notifies the receiver VM that the data is ready to be fetched from the shared memory. After successfully moving the data from shared memory to the receiver VM's kernel buffer, the correspondent shared memory slot is released and the receiver VM delivers the packet to its applications via the default kernel path.

2.3.3 Packet Analyzer and Co-resident VM Discovery

The *packet analyzer* is located in each guest VM to intercept and analyze the packets sent out by the VM. The packet analyzer can be implemented at different layers of the software stack, such as application layer, socket layer, or below the IP layer. The higher layer the packets are intercepted and analyzed, the shorter the communication path is and the better performance we can achieve. However, several serious problems may occur if the packets are intercepted at a higher layer: First, packets interception can be implemented by modifying socket APIs such as send() or sendto(), but applications need to be rewritten to incorporate these APIs, which results in poor user-level transparency [108]. Also, if packets are intercepted at socket level, the kernel network stack will be skipped, which may not be acceptable when data integrity needs to be guaranteed. For example, TCP protocol is implemented in the transportation layer to guarantee the correctness of packets transmission under unpredictable network environment. If packets are intercepted at the user level and redirected to the shared memory, then the TCP layer will be skipped, missing the detection and notice of the situations when packets got lost or corrupted during the transmission. In order to preserve all functionalities at the network stack, we implement MemPipe's packet analyzer below the IP level in the guest VM kernel.

Co-located VM discovery is responsible for determining whether a packet is delivered using MemPipe transmission path or the native inter-VM delivery path. There are two design choices for implementing co-located VM detection mechanism: *centralized* and *decentralized*. The centralized method periodically collects the status from VMs co-located on the same host and thus introduces delayed updates, which may lead to some level of inconsistency. Alternatively, the decentralized mechanism is event-driven. When a VM is deployed or migrates in or out of a host machine, the VM notifies the co-located VMs and the co-location information is updated synchronously upon the occurrence of the corresponding events. The decentralized mechanism does not require the involvement of the host and provides fresher and more consistent VM co-location information. We implement the decentralized mechanism in MemPipe.



(a) Sender VM (b) Receiver VM

Figure 5: Static shared memory allocation in co-located inter VM communication, measured by TCP_STREAM workload from Netperf, msg size = 8KB

2.3.4 Elastic Shared Memory Management

Static shared memory allocation divides the whole shared memory into equal sized partitions prior to runtime, and each partition is used for one way inter-VM communication from sender VM to receiver VM. Its advantage is easy implementation and low runtime overhead. However, its disadvantages are also obvious. First, static shared memory allocation has limited scalability. The maximum number of co-located communicating VMs is limited by the total amount of pre-allocated shared memory partitions. Second, it results in poor utilization of shared memory resources, especially when the network I/O workloads change dynamically between sender and receiver VMs, or when network communications between co-located VMs are not symmetric. For example, Figure 5(a) and Figure 5(b) show the shared memory utilization at sender VM1 and receiver VM2 respectively by executing the TCP streaming workloads generated by Netperf with message size of 8KB. By static shared memory allocation, we divide a piece of shared memory into two equal size partitions, and statically assign them to VM1 and VM2 respectively. Under this configuration, we observe that the shared memory utilization of VM1 is only around 10% while that of VM2 is as low as 0.4%, significantly lower than that of VM1. This is because compared with the ACK data sent by VM2 to VM1, the bulk data sent from VM1 to VM2 is much larger.

To address these problems inherent in static shared memory allocation mechanism, in MemPipe, we design a dynamic and proportional shared memory allocation mechanism to enable different size of shared memory to be assigned to different VMs based on the workload requirement of the applications running in these VMs.

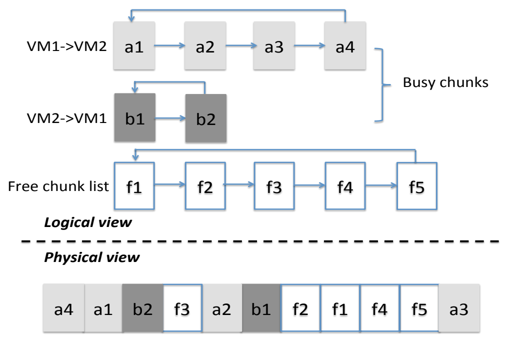


Figure 6: Dynamic shared memory management

Proportional Shared Memory Allocation. In MemPipe, the globally shared memory is organized in chunks of a system-defined size. As illustrated in Figure 6, chunks are organized in a contiguous memory address space and are allocated to or revoked from each VM in proportion to the run-time network I/O workloads between co-resident sender VM and receiver VM.

When a sender VM1 tries to communicate with a co-located receiver VM2, two shared memory pipes are established, one for VM1, denoted by VM1→VM2 and the other for VM2, denoted by VM2→VM1, as shown in Figure 6. Each VM puts its sending packets into its own shared memory pipe, while fetching the receiving data from the shared memory pipe of the correspondent VM. Memory chunks belonging to each single shared memory pipe are organized in a circular linked list, which enables the VM to treat physically separated chunks in a contiguous manner. Three categories of meta data are maintained for the shared memory pipe: *free_shm*, *shm_dscriptor*, and *chunk_descriptor*. *Free_shm* is a global meta data, which locates in the very front of the host allocated shared memory. It maintains all the free chunks in a linked list. Each *shm_descriptor* records the meta data for a single shared memory pipe − a circular linked list of busy chunks used for a sender VM to transmit data packets to a co-located receiver VM. The MemPipe kernel module in each VM checks *shm_descriptor* to get the reference of where to put the newly coming packets in the shared memory. *Shm_descriptor* also maintains the real time utilization of the shared memory pipe for dynamic allocation/deallocation. *Chunk_descriptor* is the per chunk meta data, and stores the references pointing to the start and the end of available data in each chunk and maintains the pointers to the next chunk in the same shared memory pipe.

Enlarging and Shrinking Memory Pipes. Shared memory pipes are created and maintained at each of the communicating VMs independently. Before putting a packet into the shared memory pipe, MemPipe refers to *shm_descriptor* to check whether there is sufficient free space in the current shared memory pipe to accommodate the new packet. If not, a free chunk will be added to this pipe, and at the same time, the sending VM will notify the receiving VM to fetch the packets from the pipe. The revocation mechanism is also essential to maintain the high utilization of shared memory resources and provide elastic scalability for different types of networking workloads. A separate thread is created in MemPipe to periodically monitor the usage of each shared memory pipe, and marks the chunks without any valid data as *inactive*. Although the current implementation of MemPipe allocates or removes inactive chunk from a shared memory pipe one at a time, more complex policies can be applied. For example, each shared memory pipe is initialized with a single memory chunk. Whenever it is not enough, we can add another 2^k chunks, and k refers to the number of times that the size of the shared memory pipe is detected as not enough. When the monitoring thread removes an inactive chunk from a shared memory pipe, the value k of this specific shared memory pipe is re-set to zero. This guarantees that the value of k will not keep increasing and it reflects the allocation demand of this shared memory pipe.

2.3.5 Locks in MemPipe

Active chunks in a given memory pipe are chained together using a circular linked list. To provide convenient access to free chunks for growing and shrinking the memory pipes, all free chunks are chained through another circular link, called *free chunk list*, which is accessed by the MemPipe monitoring thread in each VM.

By designing the unidirectional shared memory pipes, we reduce the number of locks required to manage shared memory resources since each memory pipe has only one writer (sender VM) and one reader (receiver VM) at any given time. This single producer-consumer pattern allows us to employ lockless design for writing and reading packets to and from the memory pipes. In addition, before the sender writes a data packet to its shared memory pipe, it always checks if the pipe has sufficient space to host the data packet. If not, it triggers the growing of the memory pipe by acquiring free chunks from the *free chunk list*. Thus, when the memory pipe is being expanded, there is no data packet being written to the pipe, and no lock is necessary in this case. Therefore, we only need two types of locks: the *free_list_lock* and the *shrink_lock*. Since all free chunks in the circular linked list are shared by multiple co-located VMs for memory pipe growing or shrinking, *free_list_lock* is required to guarantee the correctness of concurrent updates on the *free chunk list* from multiple VMs. We also need the *shrink_lock* to lock the local shared memory pipe when it needs to return some free chunks to the global free chunk list. The MemPipe monitoring thread needs to hold the *shrink_lock* before removing an inactive chunk from the correspondent shared memory pipe, which guarantees that no data will be written into the pipe when the pipe is shrinking. Similarly, MemPipe needs to hold the *shrink_lock* before writing packets data into a shared memory pipe.

2.3.6 Socket Buffer Redirection

Using shared memory pipes, we can reduce the transmission path of a network packet to four copies. In this section we introduce the technique of *socket buffer redirection* as an optimization to further reduce the number of copies by redirecting the creation of the socket buffer to the shared memory instead of inside the sender VM's OS kernel. This allows the sender VM's packets to be directly copied from the user space to the shared memory, skipping the VM kernel buffer.

The socket buffer, represented by *skb*, is the core of the network subsystem in Linux Kernel. For each network packet, both its payload and metadata are wrapped into one or multiple *skb*s. As the packet going through the network stack in Linux kernel, its header information will also be added to the *skb*. The packet will be finally sent to the wire with every necessary information placed in the *skb*.

When the socket buffer redirection is turned on in the MemPipe configuration, MemPipe will employ this optimization to further reduce the cost of data copy. Concretely, MemPipe allocates the kernel buffer pointed by *skb* from shared memory instead of from the sender VM's kernel. Thus, by calling *copy_from_user()*, this packet can be directly copied from user level buffer to the shared memory, instead of first being copied from the application buffer to the sender VM's kernel buffer and then being moved to the shared memory pipe. Note that in MemPipe, the *socket buffer redirection* (SBR) is enabled only in the sender VM's, but not allowed in the receiver VM for the sake of performance. The reason is that, if SBR is also allowed in the receiver VM, the shared memory buffer, which contains the valid packets, cannot be released until the data is successfully delivered to the user level applications.

Some previous work [44] proposed the idea of mapping the sender's whole memory address space to the receiver. Although this can reduce the cost of data copies, it may introduce unwanted risk in multi-tenant cloud where co-located communicating VMs may not be able to fully trust each other. In contrast, using our socket buffer redirection, we only allow the packet to be directly copied from the user level buffer to the shared memory of the sender VM, bypassing the copying to the sender VM's kernel buffer. Thus it does not introduce additional security risk. Also the socket buffer redirection is offered as an opt-in configuration because the implementation of socket buffer redirection requires guest VM kernel modification, though instead of directly modifying the kernel functions, we implement the corresponding functions in the MemPipe module ($\approx$100 LOC) and then hook it into the guest VM kernel when MemPipe module is inserted. Thus, the only kernel modification is adding a few hook functions.

2.3.7 Anticipation Window in MemPipe

In MemPipe, we also introduce the anticipaton window based notification grouping technique to address the performance degradation shown in Figure 2(a).

Concretely, to gain an in-depth understanding of why the performance of the shared memory approach is worse than that in native inter-VM scenario for TCP streaming workloads, we analyze the types and the amount of events occurred in the VM kernel while TCP streaming workloads are running between co-located VMs. Table 1 shows the top three most

Table 1: Top 3 most frequent invoked kernel functions

samples	image	function
4684(%7.16)	**vmlinux**	**__do_softirq**
229(%0.35)	vmlinux	csum_partial_copy_generic
222(%0.34)	vmlinux	tcp_ack

frequent kernel functions that are invoked during the TCP_STREAM workloads. We find that *__do_softirq*, *csum_partial_copy_generic* and *tcp_ack* are the top three most frequent kernel functions that are invoked during the TCP_STREAM workloads, which occupies 7.16%, 0.35% and 0.34% of the total gathered events.

The above results indicate that the frequent software interrupts incurred in shared memory based inter-VM communications under TCP_STREAM workloads are one of the main causes that severely degrade the performance of network I/O between co-located VMs. Concretely, the function *__do_softirq* is executed in a very high frequency compared with others. In Linux network subsystem, *__do_softirq* is an interrupt handler responsible for extracting packets from the socket buffer and delivering them to the applications. The CPU stack switching cost brought by executing a software interrupt handler is non-negligible, especially in the case where the frequency of software interrupt is high. Thus, reducing the frequency of software interrupts clearly offers an opportunity to further improve the performance of shared memory based inter VM communication system, especially for TCP streaming workloads.

This observation is aligned with the adaptive-tx/rx technologies which are adopted by the NIC drivers to constantly monitor the workload and adaptively tune the hardware interrupt coalescing scheme. It helps to reduce the OS overhead encountered when servicing one interrupt per received-frame or per transmitted frame. However, in MemPipe, inter VM network frames are transmitted via shared memory and never go through the hardware or virtual NICs, thus the adpative-tx/rx technologies cannot help to better manage the overheads of software interrupts in MemPipe. This motivates us to introduce *anticipatory time window (ATW)* based notification grouping to substitute the per packet notification issuing. We set the anticipation time window t to allow the sender to divide its notifications into multiple ATW based partitions of equal size N, such that each partition batches N notifications and each streaming partition is formed when N packets have arrived or when the ATW interval t expires. This ensures that the ATW based notification incurs only bounded delay by t even when there are less than N new packets in the shared memory. By tuning the parameter N and the ATW interval t, we can effectively reduce the number of notifications between sender VM and receiver VM, and significantly cut down the amount of software interrupts to be handled in both sending and receiving VMs.

2.4 Implementation Details

We implement MemPipe as kernel modules to achieve high transparency. MemPipe's functionalities are split between a kernel module running in the guest kernel and a kernel module in the host kernel. Our implementation is built on KVM[84] 3.6, QEMU[21] 1.2.0, and Linux kernel 4.1. Figure 7 gives a sketch of the MemPipe architecture that spans the host and guest VMs with an emulated PCI device to share memory between them.

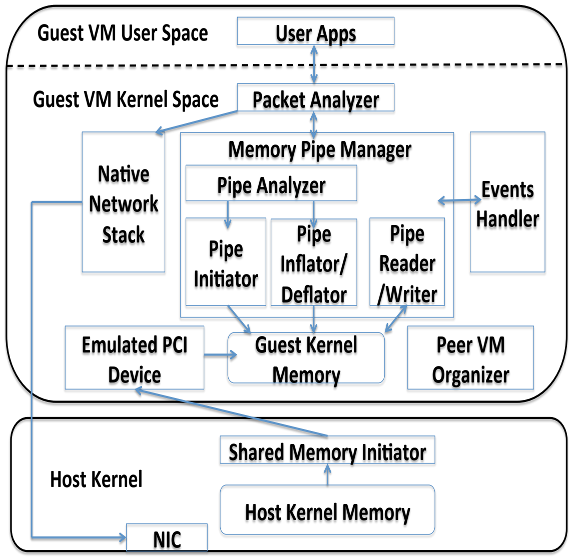


Figure 7: MemPipe implementation architecture

2.4.1 MemPipe in Host

MemPipe kernel module in the host is responsible for allocating the shared memory region from the host kernel memory and initializing the allocated shared memory region so that guest VMs are able build their own memory pipes. Concretely, the allocation of shared memory calls *shm_open()* to create a shared memory object. Then, the initiator opens a listening socket, which waits for the connections from VMs. Whenever a VMs tries to map the shared memory object to its own address space, the VM needs to communicate with the initiator in the host through this socket to get the descriptor of the shared memory object created by the host initiator. The initialization consists of two steps. First, it assigns the size of shared memory chunk based on the user configuration. A memory chunk is the basic unit that is added to or removed from a memory pipe by dynamic shared memory management. Second, the Shared Memory Initiator in the host sets up the global metadata *free_shm* that locates at the very beginning of the shared memory. *free_shm* is a data structure that contains the length and the starting address of the free memory chunk list. Whenever a VM wants to allocate more free memory chunks to its memory pipes, or remove some inactive memory chunks from its memory pipes, the VM needs to refer to this global metadata to access the free memory chunk list.

2.4.2 MemPipe in Guest

The MemPipe kernel module in a guest VM manages the shared memory pipes for its communication with other co-located VMs.

Peer VM Organizer. The Peer VM Organizer enables each VM to distinguish its co-located VMs from remote VMs (VMs running on a different host machine). We implement Peer VM Organizer by using a hashmap, with the mac address of a VM as the hash key and the value is a corresponding data structure used for establishing the memory pipe. Whenever the MemPipe in a guest VM starts to run, the Peer VM Organizer sends out a broadcast message that contains its own mac address and the mac address of the host machine it is running on top of. The other VMs receiving this message compare the host mac address in the message with the mac address of the host machine they are running on. If these two host addresses are the same, then it indicates that this message is from a co-located VM. The Peer VM Organizer in the receiving VM will add a record in its hashmap, and send a reply message back to the original VM, establishing the fact that they are co-located VMs.

Packet Analyzer. Packet Analyzer is implemented on top of Netperf to help VMs determine whether a network packet is heading to their co-located VMs or remote VMs. Netperf residing in the Linux kernel allows specific kernel modules to register callback functions with the kernel's network stack. MemPipe's Packet Analyzer registers a call back function in the VM kernel, which is responsible for intercepting each outgoing packet and retrieving the packet's destination mac address. After that, Packet Analyzer refers to the Peer VM Organizer to check whether the destination is a co-located VM. If yes, the Packet Analyzer transfers the control to the Memory Pipe Manager to establish a shared memory pipe and sends the packet out. Otherwise, the Packet Analyzer puts the packet in the native Linux network stack.

Memory Pipe Manager. Memory Pipe Manager consists of four parts: Pipe Initiator, Pipe Reader/Writer, Pipe Analyzer, and Pipe Inflator/Deflator. As illustrated in Figure 6, each shared memory pipe is exclusively used between a specific pair of co-located VMs, and it is a unidirectional pipe. Whenever a packet is identified to be sent to a co-located VM, the Pipe Analyzer will be invoked to check whether a corresponding memory pipe exists or whether the current memory pipe is sufficient for the packet. If no memory pipe is initialized, it notifies the Pipe Initiator to obtain a single free memory chunk by referring to the global metadata *free_shm*, and initialize it as the memory pipe for communication. If the current memory pipe is insufficient, additional shared memory chunk(s) will be acquired by invoking the Pipe Inflator. Once the shared memory pipe is established for hosting the data packet, the corresponding packets will be put into and fetched from the shared memory pipe by Pipe Reader/Writer. In addition, the Pipe Analyzer needs to periodically monitor the utilization of each memory pipe. If the memory pipe is under utilized, then the Pipe Deflator is invoked to dynamically shrink the pipe.

Events Handler. After putting a packet into a shared memory pipe, the sending VM notifies the receiving VM through the Events Handler to fetch the packet. In our implementation, the host machine is responsible for creating and dispatching *eventfd* objects whenever a new VM starts to run. Each VM maintains one *eventfd* object for listening to the arrival events, and multiple other *eventfd* objects for sending notifications to other co-located VMs.

Emulated PCI Device. Since KVM does not allow sharing memory between the host and the guest VM, we create an emulated PCI device in each VM to overcome this limitation. The PCI device takes the memory, which is allocated and initialized by the Shared Memory Initiator in the host, as its own I/O region. Then it maps its I/O region into the VM's kernel address, and transfers the base virtual address to the Memory Pipe Manager. Thus, the Memory Pipe Manager is able to access the shared memory from this virtual address. Although the based virtual addresses may not be the same in different VMs, they are pointing to the same physical address – the beginning of the memory allocated by the Shared Memory Initiator.

2.5 Evaluation

We evaluate the performance of MemPipe with three objectives: (1) Demonstrate MemPipe's ability to provide fast Inter-VM communications for both TCP and UDP workloads by running four unmodified micro-benchmarks; (2) Show that MemPipe's dynamic memory management delivers efficient shared memory utilization by proportional allocations with low performance overhead; and (3) Demonstrate MemPipe's effectiveness using typical network I/O applications.

Our experimental testbed consists of three physical servers: the largest one has 8 Intel Xeon 2.67GHz cores, 96GB memory and four BCM5709 Gigabit network cards machine. The other two physical machines, each has 3.0GHz Intel CPU of 4 cores, 4GB memory, two 250 GB disk and a Gigabit network interface card, all running KVM 3.6, with virtio[110] enabled, QEMU 1.2.0, and Linux kernel 4.1. We compare the performance of MemPipe with the following two conventional scenarios: (1) **Inter Machine** where native machine-to-machine network communication through physical network cards is measured; and (2) **Inter VM** where guest-to-guest network communication via standard virtualized network cards is measured.

The four ***micro-bechmarks*** used in our evaluation are: (1) **Netperf** [14], a software application to provide network bandwidth testing between two machines on a network. (2) **OSU MPI benchmarks** [16] for evaluating the performance of MPI communication, including latency and bandwidth. (3) **Netpipe-mpich** [15], a simple series of ping-pong tests over a range of message sizes to provide a complete measure of the performance of a network. (4) **Httperf** [9], a facility for generating various HTTP workloads and for measuring server performance.

We also use the following five network intensive ***user-level applications*** in our evaluation:

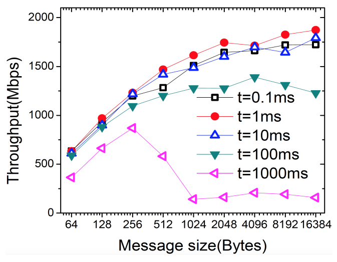
(1) **SCP (Secure Copy)**, which supports file transfers between hosts on a network; (2) **Wget** [31], a free utility for non-interactive download of files from the Web. It supports http, https, and ftp protocols, as well as retrieval through http proxies; (3)**Vsftp** [30],which stands for "Very Secure FTP Daemon", is a FTP server for Unix-like systems. (4) **Sftp** [23], a secure file transfer program.

(5) **Hadoop MapReduce** [60] applications.

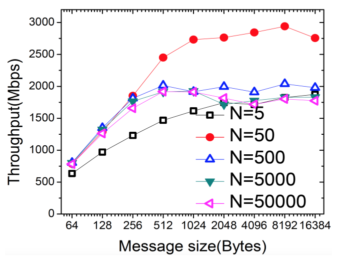
2.5.1 Performance of Anticipation Time Window

In this section we study the setting of three system parameters in MemPipe: the parameters N and t in anticipation time window based notification mechanism, which groups up to N packets within time window t, as well as the shared memory chunk size.

The first set of experiments evaluates the effect of various settings of the parameters N and t on the performance of MemPipe using Netperf [14] streaming workloads. Figure 8(a) measures the throughput of MemPipe by varying the ATW interval from 0.1 ms to 1000 ms with fixed $N = 5$. We observe that the throughput drops as the interval (t) increases and an obvious throughput performance drop is observed from 1913Mbps to 1558Mbps and 55.4Mbps, when the value of t increases from 10ms to 100ms and 1000ms respectively. In addition, we observe that the setting of $t = 1$ms represents the best case, while the setting of $t = 1000ms$ is the worst. This is because long delay may occur when the number of packets arrives within the time window t is less than N, thus all messages need to wait for t time unit before their notifications can be sent out.



(a) Throughput (N=5)



(b) Throughput (t=1ms)

Figure 8: Impact of parameters (N,t) on MemPipe

We conduct the next set of experiments by setting $t = 1ms$ and varying the ATW stream partition size N from 5 to 50,000. Figure 8(b) shows that the combination of $(N = 50, t = 1ms)$ produces the highest throughput of 2939Mbps, while all other settings result in lower throughput, especially when the message size grows bigger than 256 bytes. This is because as the message size increases, larger N will cause more packets to wait for t time unit before their notifications can be sent out. Thus, smaller value of t will lead to higher throughput. In the remaining sets of experiments, we choose $(N = 50, t = 1)$ as the default setting unless otherwise stated.

The next set of experiments is to evaluate the effect of chunk size on the performance of MemPipe. We measure the throughput of TCP_STREAM workloads with 16KB message size, while varying the chunk size from 2KB to 256KB, and observe an obvious throughput improvement from 2738Mbps to 3066Mbps, when the chunk size increases from 2KB to 8KB. However, the performance stays stable around 3000Mbps when the chunk size is larger than 8KB. Therefore, in order to guarantee a fine grained shared memory allocation, 8KB is chosen to be the chunk size of MemPipe in all subsequent experiments.

Figure 9 shows the throughput measured by TCP_STREAM workloads from Netperf to show that MemPipe has low throughput when the shared memory chunk size is 2KB compared to that of 8KB. The total shared memory is 32MB with each VM having 16MB shared memory using static memory allocation. We find that dynamic memory management with chunk size of 8KB has very similar throughput compared to that of static memory management. But dynamic shared memory management with chunk size of 2KB has consistently lower throughput for all message sizes. This is because when the data packets need more

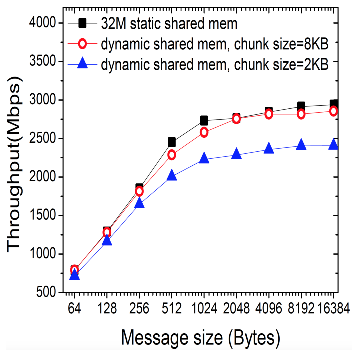


Figure 9: Performance overhead of dynamic shared mem allocation

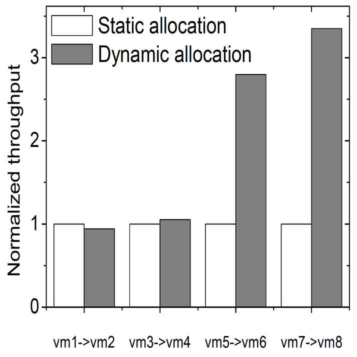


Figure 10: Impact of dynamic shared mem allocation on mixed workloads

space than the allocated shared memory pipe size, sending VM has to enlarge its shared memory pipe more frequently when using smaller chunk size.

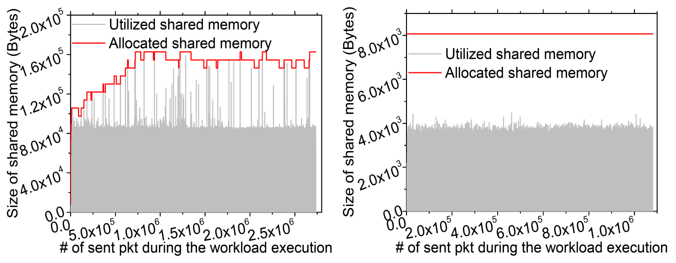
2.5.2 Dynamic Shared Memory Management

We first compares static and dynamic shared memory allocation to show that the MemPipe dynamic memory management incurs minimal and negligible overhead compared to the static shared memory mechanism, demonstrating the advantage of using dynamic shared memory allocation scheme for high performance network I/O.

The first set of experimental results is given in Table 1. We run TCP streaming workloads from Netperf with message size varying from 64 bytes to 16KB and measure allocated memory size, used memory size, shared memory utilization and throughput (Mbps). Two VMs are used for this set of experiments with one as sender VM and the other as receiver VM. For dynamic shared memory scenario, MemPipe allocates the shared memory to each VM proportionally based on the workload demand. In static shared memory allocation

Table 2: Comparison of shared memory utilization, TCP sender

msg size (Byte)	Static				Dynamic			
	Allocated(KB)	Used(KB)	Util	Throughput (Mbps)	Allocated(KB)	Used(KB)	Util	Throughput (Mbps)
64	1024	74	**7%**	762	82	75	**91%**	753
128	1024	75	**7%**	1227	88	75	**86%**	1211
256	1024	75	**7%**	1769	97	76	**79%**	1760
512	1024	81	**8%**	2283	107	81	**76%**	2258
1024	1024	92	**9%**	2652	125	90	**72%**	2584
2048	1024	99	**10%**	2986	130	99	**77%**	2950
4096	1024	100	**10%**	3215	124	99	**80%**	3146
8192	1024	100	**10%**	3381	127	99	**78%**	3322
16384	1024	100	**10%**	3438	127	100	**78%**	3402



(a) Sender VM (b) Receiver VM

Figure 11: Dynamic shared memory management in MemPipe, measured by Netperf TCP_STREAM workload, msg size = 8KB

scenario, we disabled the dynamic shared memory management in MemPipe and statically allocate 1MB of shared memory to each of the two VMs.

Table 2 shows that the sender VM is able to achieve 72%-91% shared memory utilization by using MemPipe dynamic shared memory allocation, whereas the static allocation has only 7% -10% shared memory utilization, even though the dynamic shared memory management achieves similar throughput as the static shared memory allocation. The results in Table 2 shows that the dynamic allocation in MemPipe incurs very little overhead, because the performance degradation in dynamic case stays within 2.5% compared to the static case where shared memory may be under-provisioned in one VM and over-provisioned in another.

For static shared memory allocation, when the demand of network I/O workloads on one VM exceeds the size of a statically allocated memory region, it may cause packet drops and transaction failures, even though the shared memory region on the other VM is under-utilized. In contrast, with dynamic shared memory management, as long as free memory chunks exist, the VM demanding more shared memory space can grow dynamically by adding free chunks to its shared memory pipe. In the next set of experiments, we demonstrate the benefit of dynamic shared memory management for co-resident inter-VM communication by comparing static and dynamic memory management by the inter-VM bandwidth throughput as shown in Figure 10. We setup 8VMs (VM1-VM8) on a machine with 96GB memory, and organize them into 4 sender-receiver VM pairs (VM1&VM2, VM3&VM4, VM5&VM6, VM7&VM8). All 4 pairs of VMs are communicating simultaneously. The first two pairs of VMs are running the UDP_STREAM workloads with 64B message size and the other two pairs of VMs are running simultaneously the UDP_STREAM workloads with 8192B (8KB) message size.

Figure 10 shows the achieved communication bandwidth between each pair of VMs. Although the performance between static and dynamic case are similar for workloads of 64-byte message size running on the first two pairs of VMs, MemPipe can significantly improve the throughput of the workloads with larger message size of 8192-byte running on the other two pairs of VMs. The dynamic shared memory allocation can achieve 3 times higher bandwidth throughput on average compared to the static shared memory management. This is because, with the same amount of shared memory, dynamic shared memory management is highly flexible and more adaptive. It enables MemPipe to allocate shared memory based on changing demands of workloads at runtime, i.e., allocating less memory to the workloads with small message size of 64-byte, which enables larger shared memory to be allocated to the workloads with larger message size of 8192-byte. Clearly, proportional and workload adaptive shared memory allocation can better meet the demands of different workloads compared to the static shared shared memory management.

The third set of experiments shown in Figure 11 compares the amount of shared memory that is utilized by the workloads with the amount of memory allocated in MemPipe. Figure 11(a) measures the size of the shared memory used when varying the number of packets sent during the workload execution at the sender VM. By observing the amount of utilized shared memory compared to the amount of allocated shared memory, we show that MemPipe can dynamically adjust the allocated shared memory size to meet the changing network I/O workloads at the sender VM. Figure 11(b) illustrates the shared memory allocation and utilization on the receiver VM. Based on the discussion in Section 5.1 and Figure 9, we choose 8KB as the chunk size for this set of experiments. Therefore, Figure 11(b) shows that MemPipe allocates 8KB shared memory for the VM, even though the used shared memory size is about 4KB.

2.5.3 Performance of Socket Buffer Redirection

To evaluate the effectiveness of *socket buffer redirection*, we compare the performance improvement of MemPipe over the native inter-VM communication via the traditional network stack with the following three different configurations: (1) **Shm**: VMs communicate via dynamic managed shared memory; (2) **Shm + ATW**: VMs communicate via dynamic managed shared memory, with anticipation time window enabled; and (3) **Shm + ATW + SBR**: VMs communicate via dynamic managed shared memory, with anticipation time window and socket buffer redirection enabled. The workloads are generated by Netperf with varying message sizes and Figure 12 shows the comparison results.

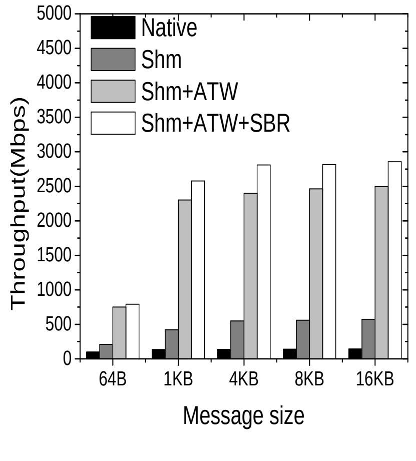


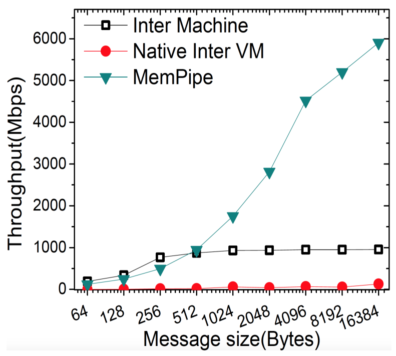
Figure 12: Performance comparison of MemPipe with three different configurations

We make two interesting observations from Figure 12. First, MemPipe shared memory management powered by anticipation time window (ATW) and socket buffer redirection (SBR) is the best configuration, which significantly outperforms the conventional approach (native) and other two configurations. On average, ATW improves the performance by 3.5 times compared with using shared memory alone. By combining with SBR, MemPipe further improves the performance by 14%. Second, the workloads with message size larger than 1KB can benefit more from the socket buffer redirection (SBR). For example, compared with the Shm +ATW configuration, Shm +ATW+SBR adds 5% throughput improvement for the workload with 64B message size but improves the performance of workload with 4KB message size by 17%. This is because the workloads with small message sizes are CPU bound, while the workloads with large message sizes are network I/O bound. With socket buffer redirection, we reduce the overhead of data transfer by avoiding the copy between the Linux kernel buffer and the shared memory. In addition, socket buffer redirection copies both the payload and the TCP/IP headers of the packets into the shared memory. Although the size of the header is similar regardless of the message size used by the workload, the relative cost brought by copying the TCP/IP headers will be smaller for workloads with larger message sizes.

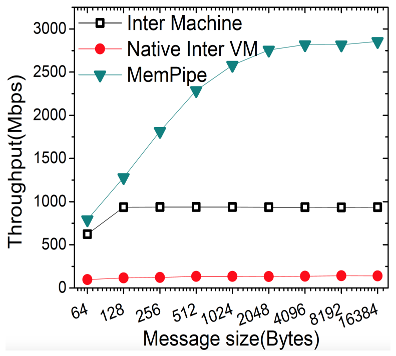
2.5.4 Performance of Micro-benchmarks

This section evaluates the performance of MemPipe by comparing it with the two conventional scenarios (Inter-Machine and Native Inter-.VM) using four different micro-benchmarks.

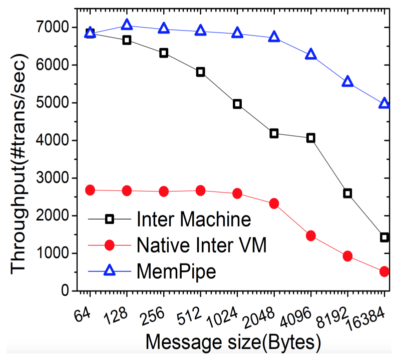
Netperf. Figure 13 compares MemPipe with two conventional network I/O approaches: inter-machine communication and native co-resident inter-VM communication. We measure the UDP and the TCP throughput in Y-axis by varying the message size from 64B to 16KB in X-axis. Streaming workloads generated by Netperf are used in Figure 13(a) and Figure 13(b), while transactional workloads generated by Netperf are used in Figure 13(c) and Figure 13(d). Each point in all four sub-figures represents the average throughput of a workload with a specific message size over multiple runs. For instance, when the message size is 16KB, the throughput of TCP streaming workload for inter-machine communication and native inter-VM communication is 936Mbps and 140Mbps respectively, while that of MemPipe is 2855Mbps.



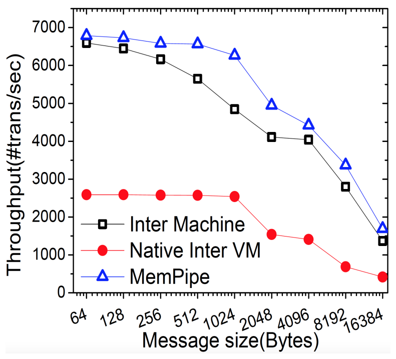
(a) UDP streaming



(b) TCP streaming



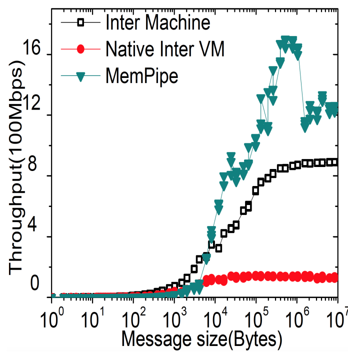
(c) UDP Transactional



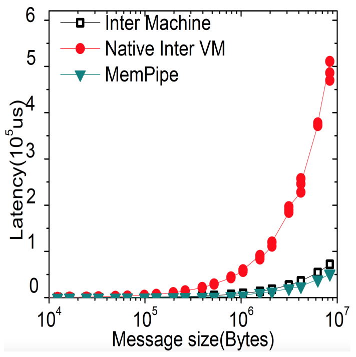
(d) TCP Transactional

Figure 13: Performance of MemPipe using Netperf UDP and TCP streaming and transactional workloads.

Figure 13(a) shows that, for UDP workloads, the throughput increases as the message size increases for all 4 scenarios. When the message size is larger than 256B, the throughput in inter-machine and in native inter-VM scenarios become relatively stable, which indicates that the network communication channel is saturated. In contrast, MemPipe consistently outperform native inter-VM scenario in all message sizes, and outperforms the Inter-Machine scenario when the message size is larger than 512B. Such performance gap increases as the message size increases from 0.5KB to 16KB with up to 32 times higher throughput compared with that in native inter-VM case. This is because for small message sizes, the performance is dominated by the per-message system call overhead. Although using shared memory, system call overhead in virtualized platform remains non-negligible compared with the Inter-Machine scenario. However, as the message size increases, the performance becomes dominated by the data transmission. The advantages of using shared memory will over-weight the overhead caused by per-message system call. Similarly, for



(a) Throughput vs. msg size



(b) Latency vs. msg size

Figure 14: Performance measured by NetPipe-mpich

TCP streaming workloads, as shown in Figure13(b), MemPipe reliably outperforms the other two scenarios for all message sizes.

Figure 13(c) and Figure 13(d) show the transaction throughput for both UDP and TCP transactional workloads respectively. For transactional workloads, MemPipe does not need to turn on ATW. We make two observations. First, the throughput of MemPipe is better than that in inter machine scenario and native inter VM scenario for both UDP and TCP transactional workloads. This is because MemPipe allows packets to be sent directly to the receiver VM through the shared memory, while in other two scenarios, packets have to go through the host OS kernel, incurring additional latency. Second, large messages take longer transmission time. Thus, the number of completed transactions decreases as the message size grows for all three scenarios.

NetPipe-mpich. Figure 14(a) and Figure 14(b) show the throughput and latency measured by NetPipe-mpich in two communicating VMs. We observe similar trend between Figure 14(a) and Figure 13(b), because NetPipe-mpich is also based on TCP protocol. The throughput achieved by MemPipe is up to 12 times higher than the native inter VM case and 1 time higher than the inter-machine case. For latency, MemPipe is as low as 73% of the inter machine case, and 11% of the native inter VM case. These observations are consistent with the results from our previous experiments using Netperf.

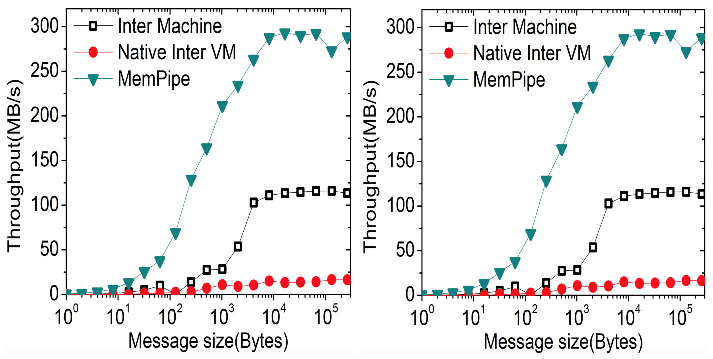
OSU MPI benchmarks.[16] We also evaluate the performance of MemPipe using the OSU MPI benchmarks. The objective of this bandwidth test is to determine the maximum sustained rate that can be achieved at the network level. The unidirectional bandwidth test,

Table 3: Related systems comparison. Table shows the performance measured by TCP&UDP streaming workloads(msg size=16KB) generated by Netperf. N/A in the tables means the approach does not support such workload or there is no such number in the paper. "Trans." means "Transparent to apps"; "Shm alloc" means "Shared memory allocation"

Approach	**Plat.**	**Setup**	**Perf. improv**	**Trans**	**Shm alloc**	ref
XenLoop	Xen	host:3.0GHzX4 CPU, 4GB RAM VM:1 core, 512MB RAM	0.8X&6.1X	yes	static	[121]
XWAY	Xen	host:3.2GHz CPU,1GB RAM VM:1 core, 256MB RAM	6.4X & N/A	yes	static	[81]
Fido	Xen	host:quad-core 2.1GHz CPUX2, 16GB RAM VM:2 cores	1.6X & 2.7X	yes	static	[44]
XenSocket	Xen	host:dual 2.8GHz, 4GB RAM	71.0X & N/A	no	static	[132]
MemPipe	**KVM**	host:3.0GHzX4 CPU, 4GB RAM VM:1 core, 512MB RAM	**20.4X & 46.0X**	**yes**	**dynamic**	.

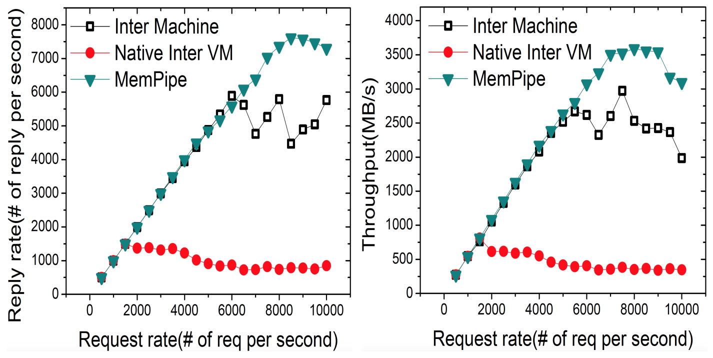
shown in Figure15(a), is carried out by having the sender sending out a fixed number of messages to the receiver and then waiting for a reply from the receiver. The receiver sends the reply only after receiving all these messages. MemPipe significantly outperforms both native inter-VM and inter-machine cases for all message sizes. Figure15(b) shows the bi-directional bandwidth test, which is similar to the bandwidth test, except that both nodes involved send out a fixed number of messages and wait for the reply. This test measures the maximum sustainable aggregate bandwidth by two VMs. As expected, MemPipe offers significantly better performance for co-located inter VM communication than that of the native inter VM scenario and slightly better than the inter machine scenario in terms of bandwidth, especially when the message size grows larger.

Httperf. We also evaluate MemPipe using Httperf. Figure16 shows that both throughput and reply rate are proportional to the request rate when the number of requests per second is low (< 2000/sec) in all the 3 scenarios. In all experiments, the request rate ranges from 500req/sec to 10000req/sec with the size of file requested at 110KB. The reason for choosing 110KB as the file size is that Httperf workloads in this case are network bounded using our experimental setup. With the increase of request rate, the performance for all three scenarios reaches the peak and starts to drop at different peak request rates. The performance of MemPipe is the best, which reaches its performance peak at the highest request rate of 8500req/sec. The native inter VM scenario has the worst performance, which begins to drop when the request rate is only 1500req/sec.



(a) Unidirectional (b) Bi-directional

Figure 15: OSU MPI benchmark test



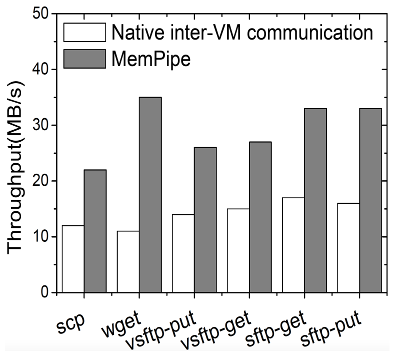
(a) Reply rate vs. req rate (b) Throughput vs. req rate

Figure 16: Httperf performance, file size = 110KB

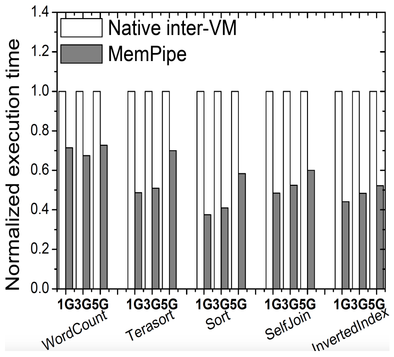
2.5.5 Performance of Network Applications

This section evaluates the performance of MemPipe by comparing it with native inter-VM scenario using a set of real-world network intensive applications. We first measure the performance of MemPipe for four widely used applications in Linux systems: ***scp***, ***wget***[31], ***Vsftp***[30] and ***Sftp***[23]. In this set of experiments, a file of 100MB is transferred 5 times from a sender VM to its co-located receiver VM, and the average throughput achieved by each application is recorded. The results are plotted in Figure17(a). Without MemPipe, the throughput measured by file transferring between two co-located VMs is about 11MB/s-14MB/s, while with MemPipe, the performance achieves 22MB/s-35MB/s.

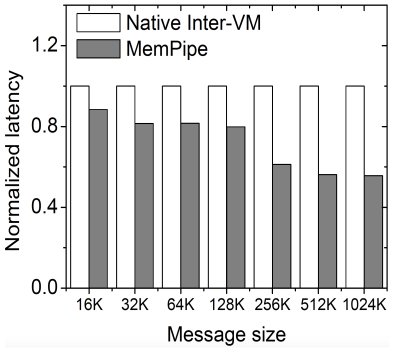
Next, we evaluate the effectiveness of MemPipe in terms of improving the performance of **MapReduce jobs**[60] and **MPI applications**[11] running in virtualized cloud. The shuffle phase is a worst bottleneck in MapReduce jobs, where each mapper needs to ship the intermediate key/value pairs with the same key to the corresponding reducers. For



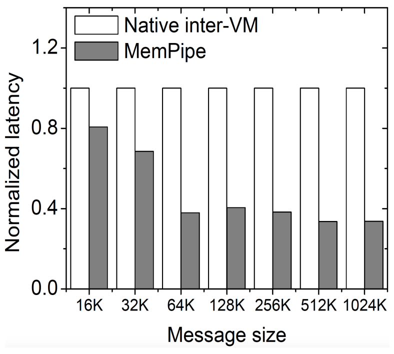
(a) Apps' throughput



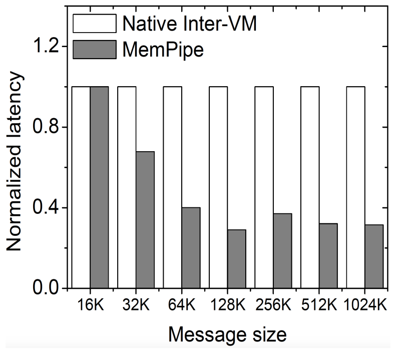
(b) MapReduce shuffle



(c) MPI Allreduce



(d) MPI Allgather



(e) MPI Alltoall

Figure 17: Effectiveness of MemPipe on applications

mappers and reducers co-located on a single host, MemPipe can significantly improving its shuffle phase performance and thus reduces the total runtime of a MapReduce job. In this set of experiments, we setup a virtualized cluster with 8 VMs on the largest physical machine. Each VM is equipped with 2GB memory and one virtual CPU. We ran six different MapReduce jobs with different size of input data (1GB, 3GB, and 5GB) in this virtual cluster. The number of map tasks is *(input data size)/(block size)* and the number of reduce tasks is configured as 8. Figure 17(b) shows that MemPipe improves the shuffle time of *WordCount, Terasort, Sort, SelfJoin, and InvertedIndex* by up to 59%. This is because all of the MapReudce jobs have large amount of intermediate data to be shuffled among the co-located VMs. For example, 5GB intermediate data needs to be shuffled for the *Terasort* job with 5GB input. Therefore, these jobs can significantly benefit by using MemPipe to increase their data shuffle efficiency. We also evaluate the performance of *MPI benchmarks* under the 8-VM setup, Figure 17(c), 17(d), and 17(e) show that MemPipe improves the latency of MPI reduce, gather, and send operations by up to 1.8, 3.0, and 3.2 times respectively compared to native inter-VM communication.

2.5.6 Comparison with Other Shared Memory Systems

Table 3 compares the performance of MemPipe with other existing representative shared memory systems. Given that XenLoop is open sourced, we measure XenLoop using our experimental setup. For *XWAY, Fido, and XenSocket*, the experiment setups and performance numbers are extracted from their papers. Given that the experimental environments and system configurations vary from one system to another, we use the normalized numbers in *Perf. improv* column to make comparison more straightforward. The two numbers represent the normalized improvement of shared memory approach over the native inter-VM for TCP_STREAM and UDP_STREAM workloads respectively. Table 3 shows that MemPipe dynamic shared memory scheme significantly outperforms XenLoop, XWay, Fido for both TCP and UDP workloads. Although XenSocket performs best for TCP_STRREAM workloads, it requires application modification and it does not support UDP_STREAM workloads.

For those systems, such as NetVM [78], ZIVM [99] and VMPI [64], they are implemented on KVM platform but they did not use Netperf workloads in their evaluation. Also all these three systems are not open sourced. Therefore we exclude them from the comparison table. However, compared to MemPipe, there are two disadvantages of these systems: First, none of them considered using dynamic shared memory management in their system design. Second, NetVM only runs on DPDK enabled platforms.

2.6 Discussion

Platform heterogeneity. The first open source release of MemPipe is implemented on KVM and written mostly as kernel modules (except socket buffer redirction). However, the design of MemPipe is generic and platform neutral. For example, Xen provides functionalities such as *grant table* and *event channel* which can be utilized to allocate shared memory and provide events delivery among VMs. One of our ongoing work is to port MemPipe onto Xen platform by utilizing these functionalities.

Extending Intel DPDK with MemPipe. Intel DPDK is a software framework that allows applications to directly poll the NIC data, thus high throughput and low latency VM network packet processing can be achieved. Also, Intel DPDK eliminates the overhead of interrupt driven packet processing in traditional OS. This is implemented by first pre-allocate a large memory region using the huge pages in Linux. The applications are then able to DMA data directly from the NIC to the memory region. However, in current DPDK, the shared memory region is current statically allocated, and its size is also arbitrarily decided. Therefore, the dynamic shared memory allocation mechanism in MemPipe can be applied to the memory region allocation in DPDK, so as to improve the utilization of the huge pages while still allowing network application performance to benefit from Intel DPDK.

Race Condition. Since the co-located VMs are automatically detected in MemPipe, VM migration may result in race condition. For example, if VM1 migrates to another physical machine right after it has been detected as a co-located VM of VM2, packets sent from VM2 to VM1 could still be put into the previous established shared memory channel between the two VMs. In this case, VM1 will not be able to fetch these packets from the shared memory because it has already migrated to another physical machine. We refer to this condition as the race condition. There are two ways to resolve this race condition. First, this problem could be relieved by periodically updating the co-locating VM list maintained by each VM. However, this approach suffers from the problem that the race condition may still occur if a VM migrates in between of two consecutive updates. In MemPipe, the following coordination mechanism is used to prevent such race condition. Before migrating, the VM needs to first freeze all its shared memory channels and consume all the remaining data in the shared memory. Then, the VM unfreezes and tears down all its current shared memory channels. Finally, the VM sends out a broadcast message to all its co-located VMs, which will update their co-located VM information list accordingly. While the shared memory channels are frozen, the other co-located VMs need to hold the packets, which are sent to this VM, and then deliver them through the native inter VM network stack when the channels are unfrozen or torn down.

Security Consideration. MemPipe adopts several mechanisms to improve the security of the shared memory pipes that are accessed by multiple VMs. For example, in MemPipe, we allocate shared memory to individual VMs and promote the principle of *"need-to-know"* by establishing unidirectional shared memory pipes from sender VM to receiver VM. This reduces the risk of shared memory to the minimum.

Another mechanism we promote is the use of VM trust group, which allows those VMs that are mutually trusted to belong to the same trust group. The shared memory channels can be established only between VMs in the same trust group. MemPipe allocates different shared memory regions from the host machine for different VM trust groups. Therefore, VMs in one group are confined by the access to the shared memory region for that group and are unable to access the shared memory used by VMs in other groups. This guarantees that packets will not be obtained by untrusted VMs. Network traffic across VM groups need to go through the native Linux network stack.

The third mechanism is to combine VM trust group with data encryption. Specifically, a key is negotiated between a pair of co-resident communicating VMs before the shared memory channel is established. All packets are encrypted in the sender VM and decrypted in the receiver VM by using the previously negotiated key. The encryption powered VM trust group is securer than using the VM trust group alone at the cost of encryption and decryption overhead.

2.7 Related Work

In the context of inter-VM communicaiton, shared memory has been studied at three different layers in the OS software stack and each has different impact on system transparency and performance overhead. Intercepting network packets at the *user libraries layer* enables a shortest communication path between co-located VMs but suffers form lacking of transparency. Representative projects in this layer include IVC [77], VMPI [64], and Nahanni [95]. Implementing shared memory based inter-VM communication at socket layer improves user level transparency but lacks of transparency to OS kernel at guest VM and host. Xway [81] and Socket-outsourcing [65] are examples in this layer. Implementing shared memory mechanisms below IP layer offers full transparency. Examples include XenSocket [132], XenLoop [121] and MMNet [106].

Recent research advances in network I/O virtualization have centered on improving the inter-VM network I/O performance by SDN and network function virtualization (NFV). Representative technology includes Intel Data Plane Development Kit (DPDK) [10] for fast packet processing using multicore systems. However, Intel DPDK has its own limitations. First, while DPDK enables applications to achieve high throughput NIC access, it is not optimized for inter VM communication. For example, if two co-located VMs are pined to different NICs, packets transferred between these VMs must go out of the host and come back via an external switch. These packets will suffer from further performance degradation because of data copy between the guest kernel buffer and host user-level buffer. Second, DPDK is not transparent to the applications in the guest VM, which means that the applications need to be rewritten and recompiled to incorporate the APIs from DPDK.

NetVM [78] is the most recent development by utilizing shared memory mechanism on top of DPDK. NetVM shows that the virtualized edge servers can provide fast packet delivery to VMs bypassing the hypervisor and the physical network interface. However, NetVM is limited to run on a DPDK enabled multicore platform and no open source is made available to date. Intel DPDK is a software framework that allows high throughput and low-latency packet processing. It allows the applications to receive data directly from NIC without going through the Linux kernel, and eliminates the overhead of interrupt driven packet processing in traditional OS. The huge pages in DPDK are statically allocated to each VM and MemPipe's dynamic shared memory mechanism not only can enhance the utilization of the huge pages but also can consolidate the edge host server's shared memory region for providing high availability and high performance network function virtualization.

2.8 Conclusions

This chapter presents MemPipe, a dynamic shared memory pipe framework for high performance communication and data transfer among co-located VMs in virtualized cloud. MemPipe employs an inter-VM shared memory pipe to enable high throughput data delivery for both TCP and UDP workloads among co-located VMs. Instead of static shared memory allocation, MemPipe manages its shared memory pipes through a demand driven and proportional memory allocation mechanism, which can dynamically enlarge or shrink the shared memory pipes based on the demand of the VMs' workloads. Furthermore, MemPipe employs a number of optimization techniques such as *time-window based streaming partitions* and *socket buffer redirection* to further improve the performance of co-located inter-VM communication. Extensive experiments on typical micro-benchmarks and network intensive applications consistently demonstrate that MemPipe guarantees flexible and efficient shared memory resource utilization and can significantly improve the performance of inter VM communication with up to 45 times of throughput increase and up to 62% latency reduction, compared with native inter VM communication for both TCP and UDP workloads.

Chapter III

IBALLOON: EFFICIENT VM MEMORY BALANCING AS A SERVICE

Dynamic VM memory management via the balloon driver is a common strategy to manage the memory resources of VMs under changing workloads. However, current approaches rely on kernel instrumentation to estimate the VM working set size, which usually result in high run-time overhead. Thus system administrators have to tradeoff between the estimation accuracy and the system performance. This chapter presents *iBalloon*, a light-weight, accurate and transparent prediction based mechanism to enable more customizable and efficient ballooning policies for rebalancing memory resources among VMs. Experiment results from well known benchmarks such as Dacapo and SPECjvm show that *iBalloon* is able to quickly react to the VM memory demands, provide up to 54% performance speedup for memory intensive applications running in the VMs, while incurring less than 5% CPU overhead on the host machine as well as the VMs.

3.1 Introduction

Cloud providers often face the challenges of both achieving high resource utilization in their data centers and at the same time allocating enough resources for individual VMs to guarantee their performance. Employing the widely adopted virtualization technology, cloud providers can multiplex a single set of hardware resources among multiple VMs, therefore increasing resource utilization by the means of overcommitting. In order to guarantee the performance of individual VMs, one simple approach is to allocate resources according to their peak demand. However, this can result in significant resource under utilization because VMs' peak demands for resources can be much higher than their average demands. Therefore, the fundamental challenge in achieving both high resource utilization and performance guarantee at the same time lies in the fact that the resources demands of VMs can vary significantly over time.

CPU and memory are the two hardware resources having the most significant impact on a VM's performance. Modern Virtual Machine Monitors (VMMs, also known as Hypervisors) typically already support dynamically allocating a pool of CPUs among different VMs. Therefore, lots of existing researches are focused on exploring how to dynamically adjust memory allocation to meet a VM's changing demand, as is the case with this chapter. Drawing from the lessons and experiences of previous work, we believe that there are two common issues need to be addressed: (1) monitoring the VM resource demand at a low cost to decide when and where to move memory among the VMs; (2) moving memory among the VMs with minimal impact on the performance of the VMs.

To address the first issue, existing researches [119] [100] [133] have proposed many methods to predict the VM memory utilization. However, an accurate prediction of VM memory working set size is still a difficult problem, especially under changing conditions [39]. Because of the fact that hypervisor lacks the knowledge of VM memory access pattern, virtualization environment makes this prediction even harder. Some researches try to break the *semantic gaps* [48] between VMs and the host by instrumenting their kernels, which brings non-negligible performance overhead [133].

The second issue has been partially achieved by the introduction of memory balloon driver [120], which allows memory to be moved among the co-located VMs and the host machine. However, balloon driver cannot work by itself. In other words, system administrators have to be involved to periodically check the memory utilization of each VM and make the decision of how to balance the memory around. There are actually some efforts to make it work automatically [2], but the system is still in its initial stage and there lacks extensive experiments to evaluate its performance. Although some researchers propose ideas to rebalance memory among VMs by using balloon driver [113] [133], they also require guest kernel modification and the overhead incurred by memory access interception in these approaches can be very high.

In this chapter, we propose *iBalloon*, which is a low cost VM memory balancer with high accuracy and transparency. No modification is required for VMs or the hypervisor to deploy *iBalloon*, which makes it more acceptable in real cloud environment. *iBalloon* runs a light-weighted monitoring daemon in each VM, which gathers the information about memory utilization of that VM. At the same time, a balancer daemon is running in the host to collects information reported by the monitor, and automatically makes the decision about how to balance the memory around VMs. The balancer finally talks to the balloon driver in the host machine to actually move the memory around. We implement the prototype of *iBalloon* on a KVM platform and the evaluation results show that with less than 5% performance overhead. *iBalloon* is able to improve VM performance by up to 54%.

The rest of this chapter is organized as following: Section 3.2 introduces the design details of *iBalloon*. Section 3.3 discuses its implementation on the KVM virtualization platform. We present our experimental methodology and explain the evaluation results of *iBalloon* in section 3.4. The related work is discussed in section 3.5 and the chapter is concluded in section 3.6.

3.2 iBalloon Design

The goal of *iBalloon* is to to keep a balanced memory utilization among VMs running on the same host while avoiding any VM from being deprived of free memory, with low cost and high accuracy and transparency. As shown in Figure 28, *iBalloon* consists of a Per-VM Monitor and a Balancer. Both the Per-VM Monitor and the Balancer are user level daemons. The Per-VM Monitor, which runs in the user space of each VM, is responsible for collecting information about the memory utilization of this VM. The Balancer, which consists of three parts: VM Classifier, Memory Balancer, and Balloon Executor, executes in the user space of the host. By using the *Exponentially Weighted Moving Average (EWMA)* model, the Balancer reads the information collected by the Per-VM Monitor, predicts each VM's future memory utilization, and makes decisions about how to rebalance the memory among the VMs. The Balancer then contacts the balloon driver to actually move memory among the VMs. Communications between the Per-VM Monitor and the Balancer are via an in memory bitmap and shared files, which are located on host and exported to the VMs by the Network File System (NFS). Since disk I/O can become a bottleneck when multiple VMs simultaneously write to the NFS directory, we put this shared directory in a memory based filesystem - *tmpfs*.

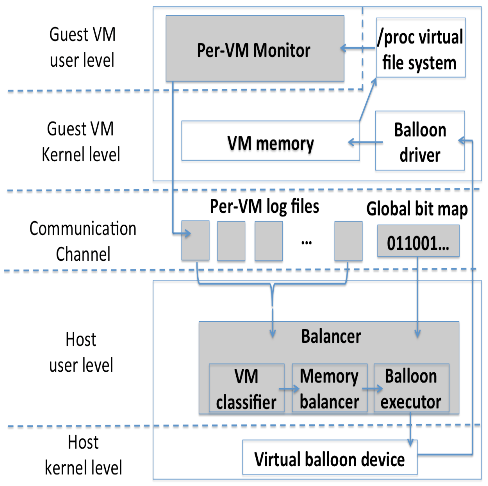


Figure 18: *iBalloon* system overview

3.2.1 Per-VM Monitor

The Per-VM Monitor is a user level daemon running in each VM. It is responsible for periodically getting memory utilization statistics from the VM and writing them into a per-VM log file, which locates in a NFS directory provided by the host and shared by all the VMs running on this host. In our design, specifically, the monitor reads two metrics from the Linux virtual filesystem /proc/meminfo: *total memory* and *used memory*, and writes them into the log file. The log file can be maintained in either an appended only or an overwritten manner. The former method keeps adding the new VM memory utilization statistics collected by the monitor to the end of the log file. The historical data in the log file may help the classifier in the host to better predict the VM's future memory usage, but the size of the log file will keep increasing with the execution of the VM, which may not be acceptable because of limited storage capacity. The latter approach always replaces the previous data in the log file with the newly collected information. In this case, the size of the log file will be small and constant, but the information provided by the file is limited. Therefore, in order to tradeoff between these two approaches, we design the per-VM log file in an overwritten manner, while keeping the aggregated VM historical memory utilization statistics in the VM Classifer running in the host. To be more specific, the per-VM log file only records two statistics: the VMs current *total memory* and *used memory*. The

VM Classifer in the host will maintain a historical information indicator for each VM, and update this indicator periodically based on the current statistics read from the per-VM log file.

Three issues needs to be addressed in order to make *iBalloon* more scalable and accurate: updating interference, monitoring frequency, and transient outliers filtering.

Updating interference. The first issue is disk bandwidth interference incurred by the Per-VM Monitor to co-located VMs. Since the monitor in each VM periodically writes the VM memory statistics into its correspondent log file, it is quite possible that multiple VMs running on the same host are writing to their own log files simultaneously. Considering the fact that disk I/O requests from all these VMs have to go through the same host machine, it will lead to severe disk contention with the increasing number of VMs running with *iBalloon*, which could in turn degrade the performance of other I/O intensive VMs running on the host. Therefore, in our design, the NFS directory exported by the host is not created from disk, but from the memory. Considering the size of each VM log file is only two integers, the additional amount of memory taken by this design is negligible even if the number of VMs is very large.

Monitoring frequency. The second issue is the monitoring frequency. As the monitor in each VM periodically collects the VM's memory utilization statistics and communicates them to the *VM Classifier* in the host, the frequency of the monitor's execution greatly affects the scalability of the *iBalloon*, especially when there are large amount of VMs running on the same host. Accurate monitoring can be achieved by allowing a high monitoring frequency, however, it can lead to high computation and communication overheads. On the contary, the data collected by a low frequency monitoring may not be accurate enough. Therefore, we employ an *adaptive frequency control* mechanism to build *iBalloon* scalable and accurate. Concretely, as described by algorithm 1, every monitor starts with a monitor interval τ. It checks whether the consecutive two monitored values vary within a pre-defined range λ. If yes, which means the memory utilization of this VM is in a relatively stable state, thus is it not necessary to update the current value in the log file. At the same time, the monitor increases the value of current monitor interval by τ. Otherwise, the monitor updates the log file with the latest value and divides the value of *interval* by half. Note that there is an upper bound as well as a lower bound for the value of *interval*, in order to prevent Per-VM Monitor from being starved or executing too frequently. We empirically set n as 10 in our evaluation.

Algorithm 1 Per-VM Monitor

```
procedure VM_MEM_MONITOR
    while true do
        $interval \leftarrow \tau$
        $interval_{max} \leftarrow n\tau$
        $interval_{min} \leftarrow \tau$
        $\gamma_{old} \leftarrow old\_vm\_mem\_util$ /*value from log file*/
        $\gamma_{new} \leftarrow get\_vm\_current\_mem\_util$
        if ($|\gamma_{new} - \gamma_{old}| \geq \lambda$) then
            $delay(\beta)$
            $\gamma_{delay} \leftarrow get\_vm\_current\_mem\_util$
            if ($|\gamma_{delay} - \gamma_{old}| \geq \lambda$) then
                $update\_log\_file$
                $interval \leftarrow MAX(interval_{min}, interval/2)$
            end if
            $interval \leftarrow MIN(interval_{max}, interval + \tau)$
        end if
        sleep $interval$
    end while
end procedure
```

Transient outliers filtering. *iBalloon* should adjust a VM's memory when the VM is indeed short of memory, which is to guarantee the correctness and stability of the VM memory management. However, short term memory burst and transient outliers are often observed in a cloud environment. Therefore, in order to prevent a VM memory from going up and down dramatically, a delay is introduced in the *iBalloon* to filter these transient outliers. Concretely, as shown in algorithm 1, when the monitor detects that the difference between the old value and the latest value is obvious enough to issue an update to the log file, it delays for a β interval, and then checks the value for a second time. If the value is still satisfy the requirement of log file update, the monitor updates the log file. Otherwise, γ_{new} will be treated as a transient outlier.

3.2.2 VM Classifier

VM Classifier is one of the *iBalloon* components that running in the host user space. It is responsible to divide the VMs running on the host into three categories based on the their predicted memory utilization. The *Exponentially Weighted Moving Average (EWMA)* model is used in the VM Classifier for the prediction. Concretely, for a specific VM, the

VM Classifier uses OFM_i to denote the VM's observed free memory, which is provided by the per-VM log file, in terms of percentage at time point i. At the same time, the VM Classifier maintains another variable PFM_i as the VM's historical information indicator. PFM_i represents the $EWMA$ of the VM memory free memory from time point 0 to i. Then, according to $EWMA$, the predicted free memory in terms of percentage at time point i is based on the value of OFM_i and PFM_{i-1}. The predicted free memory for each VM is calculated as following:

$$PFM_1 = OFM_1$$

$$PFM_i = \boldsymbol{\alpha} OFM_i + (1 - \boldsymbol{\alpha}) PFM_{i-1}, i > 1$$

in which the value of α decides whether the prediction depends more on the current observed value OFM_i or the historical information PFM_{i-1}. We set the value of α as 0.125 in our evaluation.

After calculating the predicted free memory for each VM, the VM Classifier further divides the VMs into groups based on the prediction results. Three VM groups are defined as following:

$$VM = \begin{cases} Critical & \text{if } PFM_i \in [0\%, r_1) \\ Warn & \text{if } PFM_i \in [r_1, r_2) \\ Normal & \text{if } PFM_i \in [r_2, 100\%] \end{cases} \tag{1}$$

in which, r_1 and r_2 are two values between 0% and 100% and $r_1 \leq r_2$. We empirically set $r_1 = 15\%$ and $r_2 = 30\%$ in our evaluation. The VM groups created by the VM Classifer will be fed as the input to the Memory Balancer, which then, makes decisions about how to move memory around VMs accordingly. Another parameter passed from the VM Classifier to the Memory Balancer is a VM memory array *vm_mem_old[]*, which indicates the current memory utilization of each VM.

One thing needs to be mentioned is when to trigger the VM classification. In our design, the VM Classifier maintains a bitmap, which is shared among the VM Classifer and the Monitor. Each bit in the bitmap is correspondent to a specific VM. Whenever the Monitor updates the log file, it checks whether the state of the VM (i.e. *Normal, Warn, Critical*) is changed. If true, the Monitor will set its corresponding bit in the bitmap. On the VM Classifier side, it initializes the bitmap as all zero, and periodically checks the bitmap. The VM classification is triggered whenever the bitmap is non-zero, and the VM Classifier clears the bitmap to all zero again after the classification.

3.2.3 Memory Balancer

Based on the input provided by the VM Classifier, the Memory Balancer needs to decide how to move memory around VMs so that the memory utilization of each VM can be balanced. Algorithm 2 described how the Memory Balancer work. It firstly checks whether there exist any *Critical* VMs. On one hand, if *Critical* VMs exist, which indicates that these VMs urgently need more memory, then the Memory Balancer calculates δ, which represents the total amount of memory needed to bring these *Critical* VMs to a *Cushion* level, in which VM's free memory utilization reaches 20%. Then, the Memory Balancer follows a step by step manner to decide which VMs should sacrifice their memory and how much. The ***first step*** is the calculate δ_1, which is the total amount of memory that can be taken from *Normal* VMs before making any of their free memory utilization drop to the *Warn* state. If δ_1 is already enough to satisfy all the *Critical* VMs, memory only needs to be proportionally moved from *Normal* VMs to *Critical* VMs. Otherwise, both *Normal* and *Warn* VMs need to scarifies their memory to help the *Critical* VMs, and the Memory Balancer enters ***step two***. After taking δ_1 from *Normal* VMs, all the *Normal* VMs will become *Warn* VMs. In step two, therefore, the Memory Balancer calculates δ_2, which is the total amount of memory that can be taken from all the current *Warn* VMs before making any of them in *Critical* state. In this case, the Memory Balancer should guarantee that it will not turn any *non-Critical* VM into *Critical* VM after moving the memory. Therefore, if the total available memory from *non-Critical* VMs is not able to satisfy the demand of all *Critical* VMs, the Memory Balancer will issue a "short of physical memory" warning to the system administrator. On the other hand, if there exist both *Normal* and *Warn* VMs, but no *Critical* VMs, the Memory Balancer will move memory from *Normal* VMs to *Warn* VMs to balance the memory utilization between them. Otherwise, the Memory Balancer will stay idle if there exists only *Normal* or *Warn* VMs.

Algorithm 2 Memory Balancer

```
 1: $VMs[] \leftarrow vm\_mem\_old[]$
 2: $vm\_mem\_new[] \leftarrow NULL$
 3: $vm\_mem\_delta[] \leftarrow NULL$
 4: $\Delta_i, \Delta_j, \Delta_k, \eta \leftarrow 0$
 5: $\delta \leftarrow 0$ /*total memory needed by $Critical$ VMs*/
 6: $\delta_1 \leftarrow 0$ /*maximum available memory from $Normal$ VMs before any of them dropping into $Warn$ state*/
 7: $\delta_2 \leftarrow 0$ /*besides $\delta_1$, maximum available memory from $non-Critical$ VMs before any of them drop below
    $Cushion$ state*/
 8: procedure MEM_BALANCE
 9:     if there exists any $Critical$ VM in $VMs[]$ then
10:         for each $Critical$ VM $VM_i$ do
11:             $\Delta_i \leftarrow mem\_needed\_by\_VM_i$
12:             $\delta \leftarrow \delta + \Delta_i$
13:         end for
14:         for each $Normal$ VM $VM_j$ do
15:             $\Delta_j \leftarrow mem\_available\_in\_VM_j$
16:             $\delta_1 \leftarrow \delta_1 + \Delta j$
17:         end for
18:         if $\delta_1 \geq \delta$ then
19:             $vm\_mem\_new = update(VMs[])$
20:             $return$
21:         else
22:             $set\_NormalVMs\_to\_WarnVMs$
23:             $\eta = \delta - \delta_1$
24:         end if
25:         for each $Warn$ VM $VM_k$ do
26:             $\Delta_k \leftarrow mem\_available\_in\_VM_k$
27:             $\delta_2 \leftarrow \delta_2 + \Delta_k$
28:         end for
29:         if $\delta_2 \geq \eta$ then
30:             $vm\_mem\_new = update(VMs[])$
31:             $return$
32:         else
33:             $issue\_warning$
34:         end if
35:     else if there exists both $Normal$ and $Warn$ VMs in $VMs[]$ then
36:         $average\_mem\_utilization$
37:     end if
38:     $vm\_mem\_delta[] = differ(vm\_mem\_new[], vm\_mem\_old[])$
39: end procedure
```

After all the calculations above, the Memory Balancer comes up with a new VM array *vm_mem_new[]*, which indicates the memory of each VM after the balancing. By comparing the *vm_mem_new[]* with *vm_mem_old[]*, the Memory Balancer creates another array *vm_mem_delta[]*, which represents the memory movement that should be carried out by the Balloon Executor.

3.2.4 Balloon Executor

As a user level process running in the host, the Balloon Executor receives the *vm_mem_delta[]* array from the Memory Balancer and invokes the balloon driver in the host to actually move memory around VMs. A positive number in *vm_mem_delta[]* means the memory should be added to the correspondent VM, while a negative number means this VM needs to sacrifice its memory. Since the balloon driver itself does not support moving memory directly from one VM to another, the Balloon Executor should first take the memory from one VM to the host by inflating the balloon, and then move the memory from the host to the other VM by deflating.

In Balloon Executor, a straightforward method to invoke the balloon driver in the host kernel is using the *system()* function to issue a shell command such as *"virsh qemu-monitor-command vm_id –hmp –cmd 'balloon target_mem_size"*. However, the overhead of *system()* is high since it needs to fork a child thread in order to execute the shell command. Taking this into consideration, we created our own system call *vm_balloon(u64 vm_id, u64 target_mem)* in the host to invoke the balloon driver with lower overhead.

3.3 iBalloon Implementation

We have implemented an *iBalloon* prototype in KVM virtualization platform. The two main components of *iBalloon*, the Per-VM Monitor and the Balancer, are implemented in C as user space daemons. They communicate with each other through a memory based filesystem *tmpfs*.

In order to correctly reflect the memory pressure of each VM, the collector should distinguish the memory is actually used by the system from the cached/buffered memory. The Linux operating system usually uses the free memory as cache and buffers to reduce data access latency for applications and improve the disk I/O performance. Therefore, memory used as buffers and caches should not be counted as memory that is actually used. In other words, the large amount of memory used as buffers and caches does not mean that the system memory is under a high pressure.

Besides, the Linux operating system will start swapping pages out when there is still free memory available. From our observation, for example, there are usually 120MB free memory when a VM with 1GB memory starts swapping. The reason is that Linux kernel has set a watermark for each memory zone to guarantee that the free memory of each zone will not fall below the watermark. This OS-reserved free memory is used to deal with emergency memory allocation that can not fail. Therefore, the Per-VM Monitor should consider such memory as used memory in order to accurately reflect the memory pressure in the VM.

3.4 Evaluation

In this section, we present the evaluation of *iBalloon* prototype with several widely accepted benchmarks. We begin by introducing our experimental setup. Then, we measure the performance overhead of *iBalloon*, demonstrate how mixed workloads can be benefited from *iBalloon*, and show the accuracy of *iBalloon* in terms of its VM memory prediction.

3.4.1 Experiments Setup

We conducted all experiments on an Intel Xeon based server provisioned from a SoftLayer cloud [24] with two 6-core Intel Xeon-Westmere X5675 processors, 20GB DDR3 physical memory, 1.5 TB iSCSI hard disk, and 1Gbit Ethernet interface. The host machine runs Ubuntu 14.04 with kernel version 4.1, and uses KVM 1.2.0 with QEMU 2.0.0 as the virtualization platform. The guest VMs also run Ubuntu 14.04 with kernel version 4.1. We evaluate *iBalloon* using the following benchmarks and applications:

- **Dacapo.** [4] It is a benchmark suit consists of a set of open source, real world Java applications with non-trivial memory loads. For example, some of the applications are *h2*, which executes a JDBCbench-like in-memory benchmark for executing a number of transactions against a model of banking application; *eclipse*, which executes some of the (non-gui) jdt performance tests for Eclipse IDE, and *xalan*, which transforms XML documents into HTML, etc.

- **SPECJVM2008.** [25] It focuses on the performance of the Java runtime environment (JRE) executing a single application The results reflect the performance of hardware processor and memory subsystem. It has low dependence on file I/O and includes no network I/O across machines. SPECJVM2008 includes real life applications such as *javac compiler* as well as area-focused benchmarks, such as *xml, crypto*

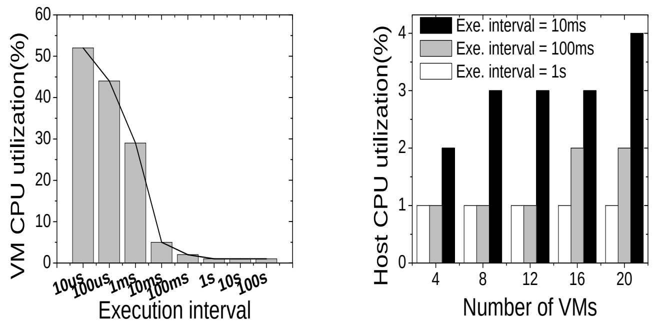
- **Himeno.** [8] It is developed to evaluate performance of incompressible fluid analysis code. This benchmark takes measurements to proceed major loops in solving the Poisson's equation using Jacobi iteration method. The performance of Himeno is especially affected by the performance of memory subsystem.

- **QuickSort.** This is a quick sort program we developed ourselves in C. We feed it with large data sets to make them memory intensive.

3.4.2 Performance Overhead

This set of experiments evaluate the performance overhead *iBalloon* incurs on VMs. In order to separate the performance overhead of *iBalloon* and that brought by the balloon driver, we disable the *Balloon Executor* in the *Balancer*, thus *iBalloon* in this set of experiments will run as usually but not actually move memory around. As we mentioned earlier, the Per-VM Monitor needs to periodically collect the memory utilization statistics from the VM's */proc* virtual file system and update its log file which locates in a memory based file system. Therefore, the performance overhead of the Per-VM Monitor could be incurred from two aspects: data collecting and log file updating. Intuitively, the higher frequently the Per-VM Monitor runs, the more CPU overhead it will incur. Although an adaptive frequency control mechanism is employed in the Per-VM Monitor, we use a fixed frequency in this set of experiments by setting a constant value σ_1 as the monitor's execution interval. Note the actual performance overhead should be no larger that what we have measured. Since according to algorithm 1, the Per-VM Monitor's execution interval will not be short than τ.

Similarly, the Balancer running in the host could also introduce performance overhead. Because every time it runs, the Balancer has to first fetch the data from multiple VM log files, then classify the VM based on a prediction based algorithm, and finally invoke the balloon device to move memory around. The performance overhead of the Balancer is related with two factors: the number of VMs running on the host, and how frequently the Balancer runs. Although the execution of Balancer depends on whether the updated information from Per-VM Monitors indicates that the state the of VM has been changed, we still set a fixed execution interval σ_2 for the Balancer in this set of experiments, to see the upper bound of the performance overhead.



(a) Per-VM Monitor overhead (b) Balancer overhead

Figure 19: Overhead of *iBalloon*

Figure 19(a) shows the VM's utilized CPU when the Per-VM Monitor is running with different frequencies. As we mentioned earlier, we vary the monitor's execution interval between different runs, but the interval is fixed for each single test. It shows that the VM's busy CPU stays as low as 1% when the Per-VM Monitor executes as frequent as every 1 second. The percentage of busy CPU begins to increase slight to 2% and 5% when the execution interval decreases to 100ms and 10ms. When the interval is shorter than 10ms, the percentage of busy CPU climbs up quickly, for example, 52% CPU is busy when the *Per-VM Monitor* execution interval decreases to 10us. From this set of experiments, we can tell that the performance overhead of Per-VM Monitor is negligible when it executes no less than every 1 second.

Figure 19(b) displays the overhead brought by the Balancer in terms of host CPU utilization. We vary the number of VMs from 4 to 20 and the execution interval of the *Balancer* from 10ms to 1s. The overhead of *Balancer* only slightly grows with the increase of its execution frequency and number of VMs. For example, when the execution interval is 10ms, the overhead increases from 2% to 4% when the number of VMs varies from 4 to 20. The overhead stays the same (1%), when the execution interval increases to 1 second. As we mentioned, since the overhead displayed in table 19(b) are supposed to be higher than that in the real case, these results indicate that the overhead of the Balancer is also negligible. Based on the experimental results above, we set τ as 5 seconds in section refmixworkloads.

3.4.3 Mixed Workloads

In this subsection, we demonstrate the effectiveness of *iBalloon* in an environment with mixed workloads by deploying it in a host with 4VMs running simultaneously. As illustrated in Figure 20, VM1 runs *Dacapo*, VM2 runs *Dacapo-plus*, VM3 runs*crypto.rsa* and *Himeno*, VM4 runs *QuickSort*. *Dacapo* includes both CPU intensive and memory intensive workloads, and they are executed sequentially in our experiments. The workloads in *Dacapo-plus* are the same as those in *Dacapo*, but are executed in a different order to create a mix of CPU and memory demand. *crypto.rsa* is a CPU intensive workload, while both *Himeno* and *QuickSort* are memory intensive. We evaluated and compare the performance of 3 cases: (1) Baseline, in which the VM memory is allocated statically; (2) Ramdisk Swap, in which a ramdisk is mounted to each VM as its swap area; (3) *iBalloon*, in which *iBalloon* is used to dynamically balance the VM memory. The VM swap area is hard disk is case the *iBalloon* case.

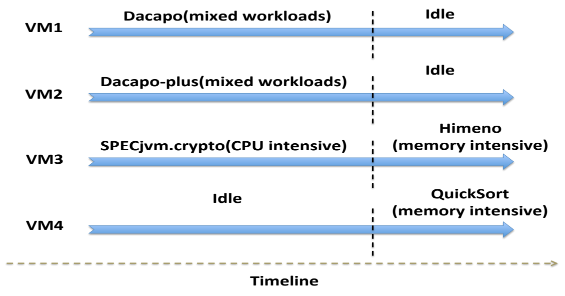


Figure 20: Mixed workload experiments setup

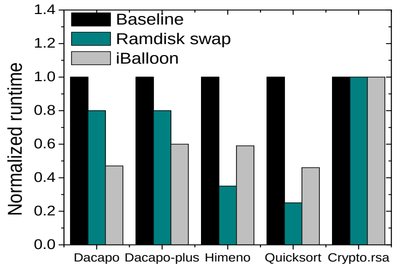


Figure 21: Normalized performance of benchmarks

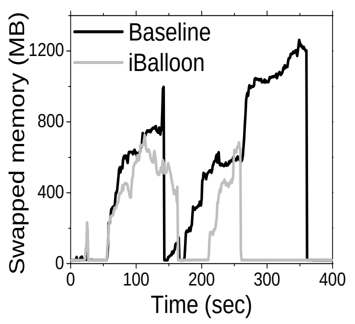
Figure 53 compares the total execution time of *Dacapo*, *Dacapo-plus*, *Himeno*, and

Table 4: Execution time of representative workloads in Dacapo and Dacapo-plus (ms)

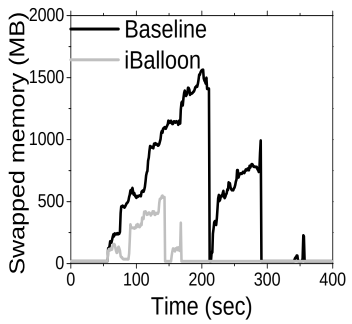
		Baseline	Ramdisk swap	iBalloon
VM1	eclipse	214,224	169,774	72,922
	h2	44,408	30,135	20,781
	jython	17,292	14,577	10,892
	fop	3,307	3,131	3,399
VM2	eclipse	168,505	124,871	93,785
	h2	36,390	26,919	13,990
	jython	12,842	12,140	11,898
	fop	3,168	3,132	3,341

Quicksort. It shows that first, with *iBalloon*, the total execution time of *Dacapo* and *Dacapo-plus* benchmarks are reduced by 52% and 50% respectively. while that of *Himeno* and *Quicksort* have been reduced by 40% and 54%, which demonstrates the effectiveness of *iBalloon* in terms of improving the performance of applications running inside VMs. Second, although using ramdisk as VM swap area can improve the application performance to some extent, it is still not as effective as using *iBalloon*. Taking the *Dacapo* benchmark for an example, its execution time in the *Ramdisk swap* cases is about 20% shorter than that in the *Baseline*, but still about 60% longer when compared with the *iBalloon* case. This is because even using ramdisk as the swap area is more efficient than using hard disk, each swap-in/swap-out operation from VM still needs to go through the block I/O layer of both the VM and the host, which will lead to much higher performance overhead compared with directly increasing the VM's memory capacity. Third, the different from *Dacapo* and *Dacapo-plus*, the performance of *Himeno* and *Quicksort* in the *Ramdisk swap* case is better than that in the *iBalloon* case. The reason is that the memory requirement of *Himeno* and *Quicksort* has exceeded the total amount of physical memory on the host, VM3 and VM4 have to swap their memory out even with *iBalloon*. Therefore swapping to the ramdisk will help more than using *iBalloon* and swapping to the hard disk.

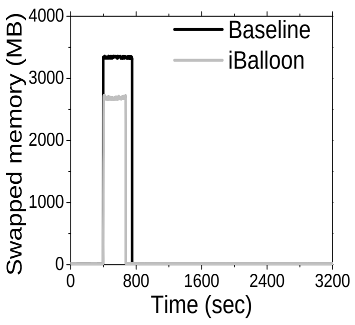
Table 4 zooms into the execution of *Dacapo* and *Dacapo-plus*, and shows the execution time of some representative workloads. We find that first, *eclipse* and *h2* have the most obvious performance improvement among all the workloads in *Dacapo* or *Dacapo-plus*. For example, in VM1, the execution time of *eclipse* has been reduced from 214224ms to 72922ms, while that of *h2* has dropped from 44408ms to 24814ms, and similar trend



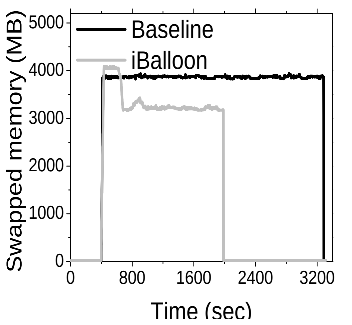
(a) VM1



(b) VM2



(c) VM3

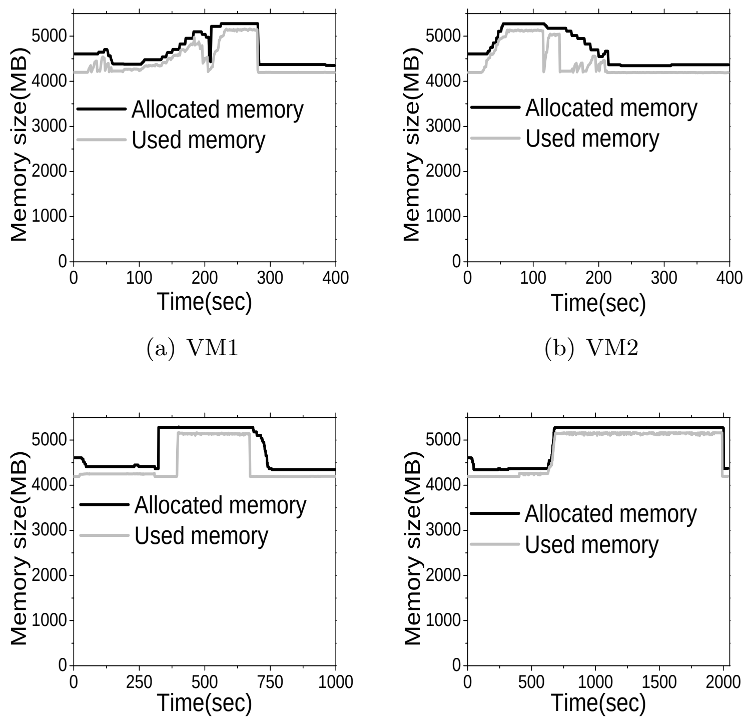


(d) VM4

Figure 22: Swapped memory in VMs

can also be observed from VM2. The reason is that *eclipse* and *h2* are the most memory intensive workloads in the suit, which result in about 580MB and 650MB memory swapping without *iBalloon*. At the same time, the execution time of some other less memory intensive workloads is slightly reduced. For instance, the execution time of *jython* reduces by 39% from 17292ms to 10892ms. An interesting observation is that the execution time of *jython* has been slightly increased in VM2 when *iBalloon* is used. This is because before *eclipse* starts to execute in VM2, memory has been moved to other VMs by the *iBalloon* to satisfy their needs. So it takes time for *iBalloon* to move memory back to VM2 when eclipse needs it.

Besides memory intensive benchmarks, a CPU intensive benchmark is also running simultaneously in VM3. Figure 53 shows the normalized throughput of *crypto.rsa*, which is the CPU intensive benchmark running in VM3 while *Dacapo* is running in VM1 and *Dacapo'* is running in VM2. It shows that the variation among the execution time of *crypto.rsa* in all cases is within 4%. This demonstrates that *iBalloon* is able to improve the performance



(a) VM1 (b) VM2

(c) VM3 (d) VM4

Figure 23: Allocated memory vs. used memory in VMs working with *iBalloon*

of memory intensive applications, while having a very low impact on that of CPU intensive ones.

Figure 22 further illustrates the amount of swapped memory in the 4 VMs at different time point during the experiment. In each sub-figure, the mount of swapped memory in cases with and without *iBalloon* are compared. It shows that the *iBalloon* significantly reduces the amount of memory pages that need to be swapped out. Take VM2 for example, the memory swapping demand lasts about 285 seconds with peak value above 1500MB in baseline. While in the *iBalloon* case, VM2's memory swapping stops at about 175 seconds, and the maximum required swapping space is between 500-600MB. For VM3 and VM4, their memory intensive workloads did not start until 400th second. *iBalloon* moves other VMs' free memory to VM3 first to satisfy its memory intensive benchmark *Himeno*. After *Himeno* finishes at around 700th second, *iBalloon* then moves the free memory to VM4 to help the execution of *QuickSort*.

Figure 23 compares the allocated memory with the amount of memory that is actually used in each VM when working with *iBalloon*. We find that *iBalloon* is able to appropriately adjust the memory size of each VM based on its workload demands, which prevents the VMs from waisting their memory resources. An interesting observation is that the allocated memory jumps up before the used memory in VM3 at 287th second. This is because right after *Dacapo* and *Dacapo-plus* finish execution, most of the memory has been moved from VM3 and VM4 to VM1 and VM2. At this point of time, *iBalloon* takes place to rebalance the memory among the 4 VMs before *Himeno* and *QuickSort* starts to run in VM3 and VM4.

3.5 Related Work

VM working set size estimation. Dynamic VM memory allocation and VM memory deduplication are the two major mechanisms that are proposed to increase the memory utilization in virtualized environment. Accurate VM memory working set size estimation is essential to the performance of dynamic memory management mechanisms.

Pin *et al.* [134] proposed using page miss ratio as a guidance of VM memory allocation. However, the tracking of page miss ratio is implemented through using a specific hardware, which is not easy to accomplish, or modifying the OS-kernel, which can results in unacceptable performance overhead.

Zhao *et al.* [133] proposed using LRU histogram to estimate the VM memory working set size. In their method, memory accesses from each VMs are intercepted by the hypervisor to build and update the LRU histograms. They introduced the concept of *hot pages* and *cold pages* to alleviate the performance cost incurred by memory access interception. But according to the evaluation result in their paper, there is still considerable performance overhead. Besides the performance overhead, accurate VM working set size prediction is difficult under chaining conditions [80] [94] [75]. Therefore, we design *iBalloon* which estimates the VM working set size via light-weighted daemons, and more importantly, makes efforts to guarantee the VM performance even if the estimation is not accurate by using shared memory swapping.

Balloon Driver vs. Memory Hotplug. In order to handle the dynamic VM memory demands and increase the memory utilization in virtualized environment, balloon driver[120] was proposed in 2002 and has been widely adopted in mainstream virtualization platforms such as KVM[84], Xen[41], VMware[109], and etc. Similarly, memory hotplug[92][113] is another technique aiming at reducing wasted memory by enabling memory to be dynamically added to and removed from VMs. Some researchers [129] explored using shared memory to increase the physical memory utilization while maintaining good VM performance. Also, there are several other works focusing on comparing balloon driver with memory hotplug from in terms of their performance and functionality.

Liu, *et al.*[92] conducted a comparative study between balloon driver and memory hotplug. They mentioned that the implementation of balloon driver is far more straightforward than memory hotplug. Since balloon driver is able to directly use the native MMU of the guest. However, balloon driver cannot enlarge the memory size of a VM beyond its *cap*, which is a preset parameter associated with each VM. Memory hogplug can go beyond the *cap*. Another finding from their work is memory hotplug should have a better scalability than balloon driver. Since balloon driver relies on the buddy system of guest MMU, which results in memory fragmentation problems. But memory hogplug can avoid this problem by adding or removing memory by a whole section.

Schopp, *et al.* [113] concisely explained how balloon driver and memory hotplug work and compared their advantages and disadvantages respectively. For example, memory hotplug allows adding memory that was not present at boot time to scale Linux up in response to changing resources, and their is no *cap* for memory hotplug to add memory. But memory hotplug has limitations on not being able to remove memory containing certain kinds of allocations. Balloon driver is able to directly use the native memory management in VM, but it could fragments the pseudophysical memory map of the guest VM.

3.6 Conclusions

We have proposed *iBalloon*, a lighted-weighted, high accurate and transparent VM memory balancing service. *iBalloon* consists of two major components: the Per-VM Monitor and a global Balancer. *iBalloon* predicts the VM memory utilization based on Exponentionally Weighted Moving Average (EWMA) model and dynamically adjust the VM memory accordingly. We evaluate the performance of *iBalloon* by using various widely accepted benchmarks and applications in a complex environment where multiple VMs running simultaneously. The results show that, with only up to 5% performance overhead, *iBalloon* is able to accurately adjust VM memory based on its real-time requirement, and greatly improve performance of applications running in the VMs by up to 54%. There are a number of extension for *iBalloon* we are considering in the future. For example, the performance of balloon driver can to be further improved. Based on our investigation, current balloon driver moves memory in a page by page manner, which may not be optimal. Batching operation could be applied to achieve better performance.

Chapter IV

MEMFLEX: A SHARED MEMORY SWAPPER FOR HIGH PERFORMANCE VIRTUAL MACHINE EXECUTION

Dynamic memory consolidation is an important enabler for high performance virtual machine (VM) execution. Ballooning is a popular solution for dynamic memory balancing. However, existing solutions perform poorly in the presence of guest swapping. For example, when the host has sufficient free memory, guest VMs under memory pressure may not be able to use it in a timely fashion. Even after the guest VM has been recharged with sufficient memory via ballooning, the applications running on the VM are unable to utilize the free memory in guest VM to quickly recover from the severe performance degradation. In this chapter, we present *MemFlex*, a shared memory swapper for improving guest swapping performance in virtualized environments. *MemFlex* is novel in three aspects: (1) *MemFlex* can effectively utilize host idle memory by redirecting the VM swapping traffic to the host-guest shared memory swap area. (2) *MemFlex* provides a hybrid memory swapping model, which treats a fast but small shared memory swap partition as the primary swap area whenever it is possible, and smoothly transits to the conventional disk-based VM swapping on demand. (3) Upon ballooned with sufficient VM memory, *MemFlex* provides a fast swap-in optimization, which enables the VM to proactively swap in the pages from the shared memory using an efficient batch implementation. Instead of relying on costly page faults, this optimization offers just-in-time performance recovery by enabling the memory intensive applications to quickly regain their runtime momentum. We evaluate *MemFlex* using a set of well-known applications and benchmarks, such as Redis, Dacapo and SPECjvm. The results show that *MemFlex* offers up to two orders of magnitude performance improvements over existing memory swapping methods.

4.1 Introduction

Virtualization is a core enabling technology for Cloud computing. By multiplexing a physical machine into virtual machines (VMs), virtualization turns the monolithic physical machine infrastructure into software-managed abstractions, providing both strong economic incentives and elegant failure isolation for applications executing on host machines. Although virtualization has shown great success in dynamic sharing of hardware resources, such as processor cores, I/O devices, dynamic VM memory consolidation remains a challenging problem for a number of reasons.

First, existing dynamic memory balancing solutions do not address the problem of guest swapping. Memory intensive applications are characterized by unpredictable peak memory demands. Such peak demand can lead to drastic performance degradation, resulting in VM or application crashes due to out of memory errors. Dynamic memory consolidation is an important and attractive functionality to deal with peak memory demands of memory-intensive applications in a virtualization platform. Ballooning is a dynamic memory balancing mechanism for non-intrusive sharing of memory between host and its guest VMs through a balloon driver with inflation and deflation operations. However, it is hard to make decisions on when to start ballooning and how much memory ballooning is sufficient. The state of art proposals typically resort to estimating the working set size of each VM at run time. Based on its estimated memory demands, additional memory will be dynamically added to or removed from the VM [42, 73, 75, 124]. However, accurate estimation of VM working set size is difficult under changing workloads [39]. Therefore, dynamic memory balancer may not discover in time that the VM is under memory pressure, or may not balloon additional memory fast enough. As a result, the VM under memory pressure may see more guest memory swapping events and more drastic performance degradation.

Second, by virtualization design, when a virtualized host boots, it treats each of its hosted VMs as a process and allocates it a fixed amount of memory. Each guest VM is managed by a guest OS, independently (and unaware of the presence) of the host OS. Thus, even when the host has sufficient free memory, the guest VMs under memory pressure are unaware. Thus, any delay in dynamic memory balancing can cause the VM not be able to utilize the host idle memory in a timely fashion. As a result, VMs under memory pressure will experience increased guest swapping, which can lead to VM and applications to crash due to high latency induced timeout.

Finally, we observe through experiments that the applications running on the VM are often unable to utilize the free memory that have been inflated to the VM fast enough due to poor performance of VM swapping-in operations.

In this chapter, we argue that (i) efficient guest VM swapping can significantly alleviate the above problems and (ii) fast VM swapping is a critical component to ensure the just-in-time effectiveness of dynamic memory balancing. We design and implement *MemFlex*, a host-coordinated shared memory swapper for improving VM swapping performance in virtualized environments. *MemFlex* can effectively utilize host idle memory by leveraging a shared memory swap area between each VM and its host. There are a number of challenges for redirecting guest VM swapping to the host-guest shared memory area. First, the shared memory should be organized in a way that allowing multiple VMs to shared dynamically and access concurrently. By default, the OS of every VM tracks where the pages are swapped out. In such as shared environment, *MemFlex* needs to correctly and effectively track the swapped out pages from multiple VMs. Second, since there could be no more space in the shared memory to accommodate the swapped out pages from the VMs, *MemFlex* needs to enable VMs to interact with both shared memory and the traditional swap devices, such as disk partitions or files, at the same time. Third, in order to achieve good utilization of the shared memory, a VM needs to proactively release its currently used shared memory as soon as it gains enough free memory from the host.

With these challenges in mind, we design *MemFlex*, a flexible shared memory swapper with three original contributions: (1) By redirecting the VM memory swapping to the host-guest shared memory swap partition, *MemFlex* avoids the high overhead of disk I/O for guest swapping as well as guest-host context switching, and enables the guest VM to respond fast to the newly ballooned memory and to quickly recover from severe performance degradation under peak memory demands. (2) To handle the situation of limited shared memory swap area due to insufficient available memory at the host, *MemFlex* provides a hybrid memory swapping model, which treats shared memory swap partition as the small primary swap area and the disk swap partition(s) as the secondary swap area. This model enables fast shared memory based VM swapping whenever it is possible and a smooth transition to the conventional guest OS swapping on demand. (3) To address the problem of slow recovery of memory intensive workloads even after sufficient additional memory has been successfully allocated via ballooning, we provide a fast swap-in optimization to proactively swap-in the pages resident in the shared memory swap area, reducing the high cost of frequent paging based swap-in.

We implement the first prototype of *MemFlex* on KVM platform and evaluate the performance of *MemFlex* using a set of well-known applications and benchmarks, such as Redis, Dacapo and SPECjvm. We show that *MemFlex* significantly improves the VM execution throughput by up to 3x and reduces the application runtime by up to 70%, when the host has sufficient free memory but some of its VMs are under high memory pressure. More importantly, even with a small amount of shared memory swap area. *MemFlex* with *proactive swap-in* achieves two orders of magnitude performance improvement on VM memory swap-in after memory recharging via ballooning, and enables both VM and application execution to quickly regain their runtime momentum.

The rest of this chapter is organized as follows: We review the related work in Section 4.2 and motivate our work with experimental observations in Section 4.3. We describe the design and implementation of *MemFlex* in Section 4.4, report our experimental evaluation results in Section 4.5 and conclude the chapter in Section 4.6.

4.2 Related Work

The state of art research on improving VM execution efficiency for memory intensive applications is centered on dynamic memory balancing. Most of existing efforts have been dedicated to developing different host-guest coordination mechanisms along three threads.

Host coordinated ballooning and host swap. The first thread is to introduce host coordinated ballooning and host swap. The balloon driver, proposed in 2002 [120], has been widely adopted in mainstream virtualization platforms, such as VMware [109], KVM[84], Xen[41]. Most of them embed a driver module into the guest OS to reclaim or recharge VM memory [39]. A fair amount of research has been devoted to periodic estimation of VM working set size because an accurate estimation is essential for dynamic memory balancing using the balloon driver. For example, VMware introduced statistical sampling to estimate the active memory of VMs [120, 29]. Alternatively, [133] builds and updates the page-level LRU histograms by having the hypervisor intercepting memory accesses from each VM and uses the LRU-based miss ratio to estimate VM memory working set sizes. [134] proposed to implement the page-level miss ratio estimation using specific hardware to lower the cost of tracking the VM memory access. However, [80, 94, 75] show that accurate VM working set size prediction is difficult under changing conditions.

Host swap is a guest OS transparent mechanism [29] to deal with the shortage of host free memory by having host or hypervisor swapping out some inactive memory pages to host-specific disk swap partition, without informing the respective guest OS. However, such uncooperative host swapping can cause some serious problems between host OS and guest OS, such as double paging [39]. VSwapper [39] tracks the correspondences between disk blocks and guest memory pages to avoid unnecessary disk I/O caused by uncooperative memory swapping between host and guest VMs. It is a disk-based VM swapping facility to reduce the unnecessary guest and host swapping, but VSwapper does not improve the VM page-level swap performance.

In contrast, *MemFlex* is a shared memory swapper with hybrid swap-out and proactive swap-in optimizations. It is designed for fast guest swapping and for accelerating application performance recovery immediately after ballooning. Also *MemFlex* is complimentary to VSwapper and can leverage VSwapper and page deduplication mechanisms, such as Singleton [117] and Difference Engine [73], to reduce the amount of unnecessary or inconsistent paging.

Coordinated memory management. The second thread is centered on redesigning operating system (OS) to enable more efficient host-guest coordination. The transcendent memory (tmem) on Linux by Oracle and the active memory sharing on AIX by IBM PowerVM are the two representative efforts. For example, transcendent memory [96] allows the

VM to directly access a free memory pool in the host. Frontswap [6] is a Linux kernel patch that using the tmem as a VM swap space, and it is currently working on Xen. In Frontswap, a hypercall has to be invoked for each swapped out page, and the swap-in operations depend on the page faults, which are quite costly. *MemFlex* is running on KVM, and it avoids the hypercall and page fault for the page swap-out and swap-in respectively. [85] shows that this pool of memory can be used by Guest OS to invoke the host OS services and by the host OS to obtain the memory usage information of the guest VM. [112] allows the applications that implement their own memory resource management, such as database engines and Java virtual machines (JVMs), to reclaim and free memory using application-level ballooning. However, most of the proposals in this thread rely on some serious changes to guest OS or applications, making it harder for wide deployment of the solutions.

Dynamic memory consolidation. The third thread is centered on complimentary techniques to improve dynamic memory consolidation, ranging from memory hotplug, collaborative memory management to remote memory swapping. Memory Hotplug [92, 113] was proposed to address the problem of insufficient memory or memory failing at both guest VMs and host. It refers to the ability to plug and unplug physical memory from a machine [92] without reboot to avoid downtime. [92, 113] conduct comparative studies between balloon driver and memory hotplug, commenting that the implementation of balloon driver is more straightforward than memory hotplug. Collaborative memory management [114] proposed a novel information sharing mechanism between host and guests to reduce the host paging rate. In addition, several efforts have engaged in combining swapping to the local disk with swapping to the remote memory via network I/Os: [63] organizes the memory resource from all nodes of a cluster into one or more memory pools that can be accessed via high speed interconnect. It deals with memory overload by swapping the VM memory pages to network memory, and introduces an optimization to bypass the TCP/IP protocol by removing IP routing to improve the performance of swapping to the cluster-wide remote memory. Overdriver [123] distinguishes guest swapping caused by short term overloads from those due to long term overloads by detecting the reason of guest swapping. It uses the remote memory when guest swapping caused by short term overloads and migrates VMs if the swapping is instigated due to long-term overload. [128] proposed a synthetic peer to peer swapping mechanism by organizing the host free memory from a set of machines into a distributed virtual pool of swap memory. Although each of these existing proposals has shown some performance improvement for a selection of workloads, the remote memory swapping solutions have some limitations: (i) Network I/Os in Cloud data centers are known to be expensive and highly unpredictable. (ii) Swapping to remote memory needs to pay additional cost for routing. (iii) Network bandwidth and network I/O delays are beyond the control of local host.

In comparison, *MemFlex* promotes to use a small amount of host-coordinated shared memory as the fast swap partition and resort to conventional secondary storage based swap partition when there is insufficient free memory on the host. It is orthogonal but complimentary to using network memory for guest swapping. To the best of our knowledge, *MemFlex* is the first shared memory swapper with host coordinated two level memory swapping and fast proactive swap-in optimization.

4.3 Motivation and Overview

Can guests swap when host memory is underutilized? Guest swapping refers to memory paging in guest VM when its memory demand exceeds its allocated memory. In virtualized environment, guest swap can happen even when there is enough free memory on the host. To demonstrate this, we conduct a set of experiments on an Intel Xeon server with 24GB memory available. We create VM1 with 4GB DRAM and 4GB disk swap area. We first run a Redis server on VM1, which is loaded with 3GB data in memory. We measure the throughput performance of the Redis [22] server by running a YCSB workload [53] on a client machine, which consists of 50% uniform update and 50% uniform read operations on the data in the Redis server. In many big data applications, the 50% mix of read and write workload is considered more representative compared to the read most or the write most workloads. The reason that we choose uniform read and uniform write is to reflect the workloads of unpredictable read and write operations rather than pre-defined read and write workloads. After the workload is running for 10 seconds, we launch another memory intensive application *memAlloc* at the 11th second on the same VM1 to allocate 3GB memory. The VM memory utilization is monitored at an interval of 5 seconds and the balloon driver will be triggered if necessary to give more memory to VM1.

Figure 24 shows that the throughput of Redis server is increasing during the first 10 seconds and there is no guest swap. However, as soon as the *memAlloc* starts on VM1, the Redis server starts experiencing drastic throughput degradation due to increased guest swapping.

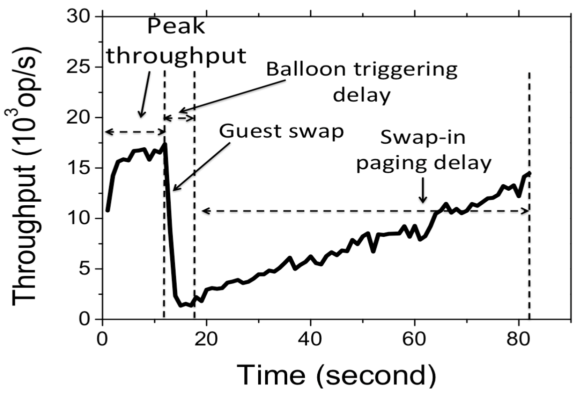


Figure 24: Delays in Dynamic VM memory balancing

Is guest swapping negligible when dynamic memory balancing is active? Existing dynamic memory balancing solutions such as ballooning may suffer from three types of delays: (i) the timing delay for triggering ballooning, (ii) the balloon driver delay for moving sufficient memory from host to guest VMs, and (iii) the VM swap-in paging delay. We first use the same set of experiments in Figure 24 to illustrate these three types of delays. In addition to the delay in detecting the need for ballooning, we observe that after successful allocation of an additional 3GB memory to VM1, with a total of 7GB memory on VM1, the Redis server still cannot immediately recover to the peak throughput it had before *memAlloc* started. It takes quite some time (more than 60 seconds in this case) for the Redis server to slowly recover. We argue that the guest swapping performance is critical to the effectiveness of dynamic memory balancing and should be addressed as an integral part of a dynamic memory balancing solution.

Does guest swap due to high CPU utilization?

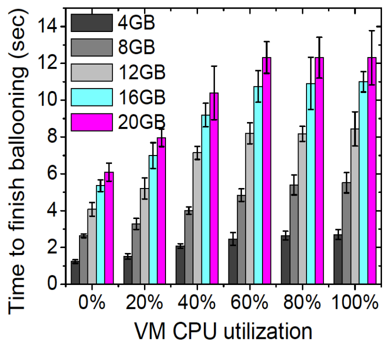


Figure 25: Internal delay

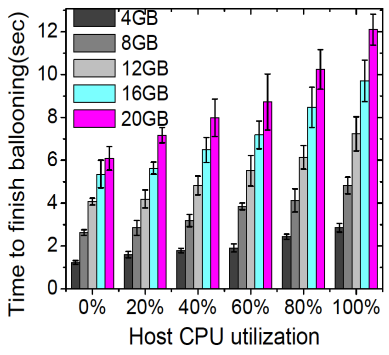


Figure 26: External delay

We observe that when CPU utilization on the host and its VMs is high, the balloon driver will suffer from longer delay in inflating/deflating memory, which can further increase VM swapping. We first show that the performance of the balloon driver can be affected in the following scenarios: (1) when the VM with high memory pressure also experiences a high CPU utilization, and (2) when the other VMs on the same host are running CPU intensive workloads. We call the delays in (1) and (2) the *internal delay* and the *external delay* respectively. We set up VM1 on a KVM host and create workloads on VM1 with CPU utilization varying from 0% to 100%. We measure how long it takes the balloon driver to move a specific amount of free memory from the host to VM1 (4GB to 20GB). Figure 25 displays the elapsed time, which includes the CPU time consumed by the balloon driver and the waiting time for the balloon driver to be scheduled by the CPU scheduler. Clearly, the elapsed time of the balloon driver increases as the CPU utilization on the VM is getting higher.

We next measure the impact of external delay. We set up different number of CPU

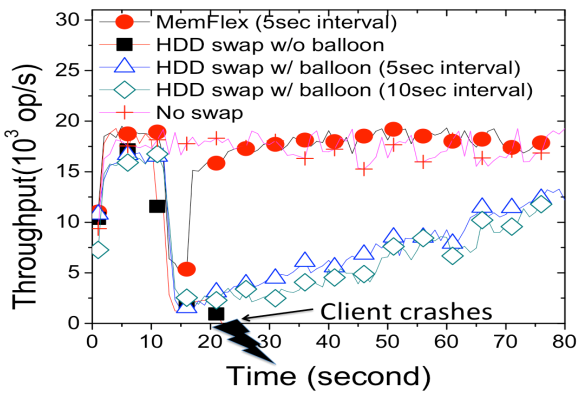


Figure 27: Performance of Redis memory intensive workloads

intensive VMs on the same host such that the overall CPU utilization of the host can be varied. Figure 26 displays the elapsed time of the balloon driver under varying CPU utilization rates while it moves 4GB to 20GB of free memory from the host to the VM. When the host CPU utilization is 80%, it will take the balloon driver up to 12.3 seconds to move 20GB memory, compared to three seconds to move 4GB memory and about eight seconds to move 20GB when the host CPU utilization is 40%.

In the presence of both internal delay and external delay, the high CPU utilization in both VM and the host will introduce even longer waiting time and thus longer total elapse time for the balloon driver. This long latency of the balloon driver (memory balancer) will further intensify the guest memory swapping in the VM, because the VM cannot release its memory pressure in time.

How can *MemFlex* shared memory swapper help? *MemFlex* development is motivated by the observations that VM memory swapping happens (i) when the host memory is underutilized, or (ii) when there are delays in dynamic memory balancing, or (iii) when the dynamic VM memory balancing mechanisms, such as ballooning, are unable able to satisfy the memory demand of the VMs quickly enough, especially under high host and VM CPU utilization. Before presenting the complete design of *MemFlex* in the next section, we use an example to illustrate how much performance improvements one can expect from the *MemFlex* shared memory swapper.

The experimental set up is the same as we mentioned in Figure 24. We compare the

Redis server performance for the following five cases: (i) *MemFlex*, (ii) HDD swap without ballooning, in which the VM memory is swapped to the disk, (iii) HDD swap with ballooning at 5 seconds interval, (iv) disk swap with ballooning at 10 seconds interval, and (v) no swap, in which the VM is initialized with sufficient extra memory such that no swap will occur under the same experiments. VM memory utilization is monitored at a specific interval and the balloon driver is triggered when necessary.

In the *HDD swap* cases, 3GB memory is moved from the host to VM1 when its memory is under pressure, while in the *MemFlex* case, 2GB memory is moved from the host to VM1 while the other 1GB is reserved as its swap area. Figure 27 shows the results. We observe that even though VM1 has 7GB memory after the balloon driver finished moving of 3GB memory, which is enough to run both the Redis server and the *memAlloc*, the throughput of Redis recovers very slowly in the two cases of HDD swap with ballooning over the period of the next 60 seconds. This is because when VM1 needs to swap in the memory pages residing in the HDD swap partition on demand, it incurs costly page faults. In the case of HDD swap without ballooning, Redis server crashed due to high memory pressure induced timeout. In contrast, *MemFlex* can quickly respond to the additional memory added to VM1 and provides just in-time performance recovery for the application.

We use the interval setting of 5 seconds and 10 seconds in Figure 4 for triggering the Ballooning driver. In practice, the interval of ballooning is typically determined by the system administrator based on tradeoff between the VM execution performance and the overhead of estimation and balancing of VM memory. For example, the VMware ESX Server balances the VM memory every 30 seconds [120]. The larger the ballooning (balancing) interval is, the slower for the dynamic memory balancer to respond by ballooning more memory to a VM with high memory pressure, the heavier the memory swap traffic will be and the sharper performance degradation that this VM may experience. We choose a small interval setting of 5 seconds in Figure 4 to show that MemFlex can respond effectively even under a much shorter balancing interval. We have developed intelligent ballooning facility, called iBalloon [130], a light weight dynamic ballooning monitoring service to help system administrators to determine this interval parameter, with the goal of just-in-time memory balancing.

4.4 MemFlex

The goal of *MemFlex* is to design and implement a shared memory based high performance guest VM swapping framework. Figure 28 gives a sketch of the *MemFlex* system architecture. Instead of swapping to the disk, *MemFlex* intercepts and redirects the swap-out guest memory pages to the guest shared memory swap space within the host memory region. This minimizes the high overhead of disk I/Os and alleviates the problem of uncooperative swapping. The shared memory swap partition is organized into multiple shared memory pools, one per VM, and each corresponds to a host-guest shared memory area. This design presents a clean separation and facilitates the protection of the shared memory swap area of a VM from unauthorized access by the other VMs on the same host. In addition, *MemFlex* shared memory swapper supports the following two features:

- **Hybrid swap-out.** When the amount of shared memory is not able to hold all the memory pages swapped out from a VM, *MemFlex* uses the shared memory swap area as the fast primary swap partition and automatically resort to secondary storage swap partition for least recent swap pages. This feature improves the overall guest swapping performance even when there is insufficient shared memory resource.

- **Proactive swap-in.** Once a VM is recharged with sufficient free memory via the balloon driver, it needs to swap-in the pages from the shared memory swap area back to its main memory. *MemFlex* improves swap-in performance from two aspects: (i) the overhead of swap-in is reduced from disk I/O to page table operations for pages resident in the shared memory swap area, and (ii) instead of demand paging from the guest VM, *MemFlex* enables VM to proactively swap in the pages from the shared memory by minimizing the costly page faults. This optimization effectively addresses the problem of slow recovery of application performance upon the addition of sufficient memory to the VM under pressure.

4.4.1 Host Memory based VM Swapping

There are two alternative ways to implement the host-coordinated memory swapping: (i) swapping to host ramdisk and (ii) swapping to host-guest shared memory. The ramdisk based approach is simple and requires no guest kernel modification, but have higher VM swap overhead, whereas the shared memory based approach delivers better performance with a small change to the guest kernel.

Ramdisk based Swapping. Ramdisk based swapping improves efficiency of traditional VM swapping by swapping to host memory instead of the VM disk image. This is achieved by building a ramdisk from the host memory and mounting this ramdisk as the swapping area of the guest VM. Each ramdisk can be mounted to a different VM at different times based on the swapping demands of the guest VMs. There are several advantages with swapping to ramdisk, compared to swapping to the VM disk image. First, it makes the VM memory swapping operation much faster, since the swapping daemon no longer needs to access the physical disk. Thus, the performance of memory intensive applications in the guest VMs can be improved. Second, since the VM memory swapping traffic is redirected to the host memory, it minimizes the disk I/O bandwidth consumption and alleviates the disk I/O performance interference incurred by memory intensive applications. The third advantage is that setting up the ramdisk-based swapping is transparent to the applications, the guest VMs and the host OS. No modification is required.

However, ramdisk-based swapping has some inherent performance limitations. Although the ramdisk is residing in the host memory, it is still used by the guest VM as a block device. From the perspective of guest VMs, writing to the ramdisk is exactly the same as writing to its disk though faster. As a result, certain overheads that are applicable to disk I/O will also apply to ramdisk I/O. Concretely, a disk I/O request from the application running on a guest VM will first go through the guest OS kernel before the request and the I/O data are copied from the guest kernel to the host OS kernel; this represents a data transfer between the host user space and the host kernel space. Hence, when a guest VM swaps its memory pages into a mounted ramdisk, there are two major sources of overhead: the frequent context switch and the data copying between the host user space and host kernel space. We use

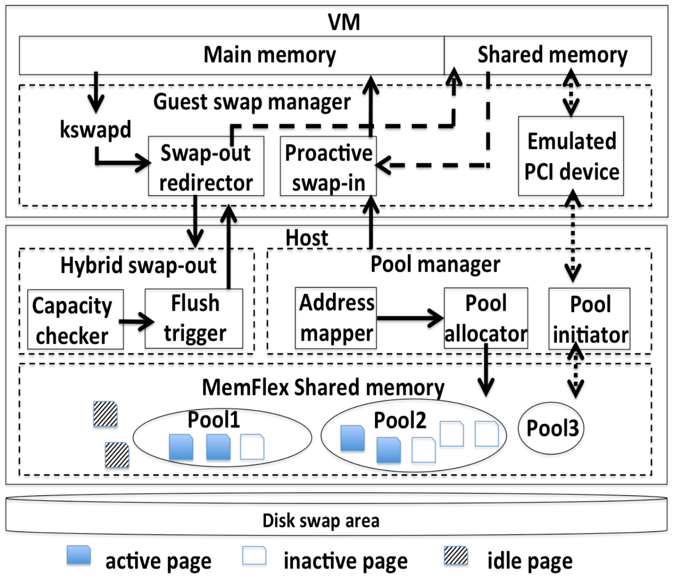


Figure 28: MemFlex system overview

Perf tool [19] to evaluate the performance overhead of ramdisk-based swapping by letting a VM swap 512MB memory into a ramdisk provided by its KVM host. It shows that (i) 7.86% CPU cycles are spent on *copy_user_generic_string()*, which is a kernel function that copies user generated data into kernel space; and (ii) 7.50% CPU cycles are spent on context switching between the guest VM and the host. Although ramdisk based swapping may outperform the traditional disk-based swapping (baseline), these overheads limit the efficiency of ramdisk based swapping when compared to the *shared memory based swapping* (see detail in Section 4.5).

Shared Memory based Swapping. *MemFlex* is a shared memory based swapping approach. Instead of mounting a host resident ramdisk to the guest VM, in *MemFlex*, a memory region provided by the host is mapped into the guest VM address space and used as the host-guest shared swapping area for the guest VM. Figure 28 illustrates the workflow of shared memory based swapping implemented on the KVM platform. As a part of the system initialization, A shared memory region with a pre-configured size is initialized. The shared memory area is divided into multiple adaptive *pools*, and each pool is corresponding to a specific VM. A pool manager is working in the host, which has two functionalities: (i) maintaining the mapping between the page offset in the VM swap area and the address in the corresponding shared memory pool, and (ii) dynamically adjusting the size of each VM pool. The pages in the shared memory are categorized into three types: *active* pages refer to ones being used by the VMs as their swapping destination, *inactive* pages are those allocated to some pool but has not being used yet, and *idle* pages indicate those shared memory pages that do not belong to any pool, as shown in Figure 28. The *kswapd* is a default kernel daemon, which is responsible for memory page swap-in and swap-out. The swap redirector intercepts and redirects the swap in/out pages from/to the VM shared memory area. The proactive swap-in handles both on-demand swap-in and batch swap-in.

Compared with swapping to a block device, such as hard disk or ramdisk, the shared memory swapping mechanism in *MemFlex* has a number of advantages. *First*, by intercepting all the memory swapping traffic in the guest VMs and redirecting them to the shared memory regions between the host and its guest VMs, both VM and host file system can be skipped in VM memory swap and no block I/O needs to be carried out. Since the shared memory has mapped into the guest VM's own address space, additional memory address translation can also be avoided. Also, *MemFlex* performs within a single address space and avoids costly guest-host context switching.

Second, considering that the baseline VM memory swapping will generate block device I/O traffic, which can make a memory intensive VM also become a disk I/O intensive VM. Therefore, when memory intensive VMs are running together with disk I/O intensive VMs, severe disk I/O interference will degrade the performance of memory intensive and disk I/O intensive VMs. By eliminating or minimizing the disk I/O traffic caused by VM swapping, *MemFlex* also helps alleviate the performance interference between memory intensive VMs and disk intensive VMs.

Third, with *MemFlex*, the cost of double paging can be largely reduced. Even though double paging may still happen in the presence of host swap in *MemFlex*, its cost will be reduced from two block device I/O operations (one by reading the page from the host swap area to the VM memory, and the other by writing the page from the VM memory to the VM swap area) to only one block device I/O operation, which is the cost of a regular VM swap-in.

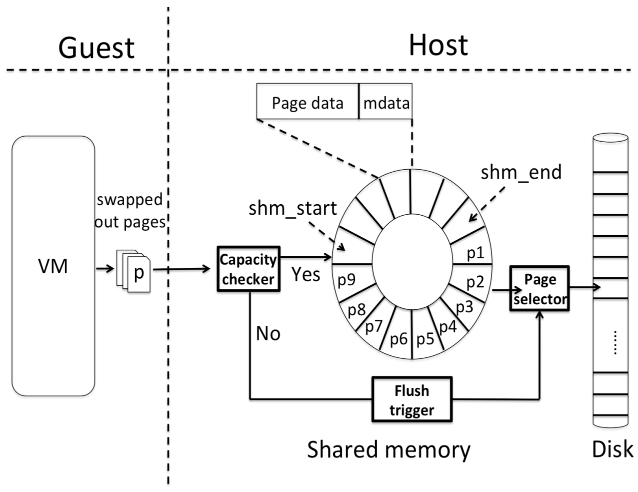


Figure 29: Hybrid (disk and shared memory) swap-out.

4.4.2 Hybrid Swap-out

Hybrid Swap-out is designed to handle the situation when the size of shared memory is not large enough to hold all the swapped pages from a VM. To enable the shared memory as the fast primary swap partition and smooth transition to the disk swap area as the secondary choice, *MemFlex* employs a hybrid guest swap model by using both shared memory and the disk. VM swapped out pages that cannot be kept in the shared memory will be written to the VM disk swap area.

There are a number of strategies to design the hybrid swap-out model. For instance, if we maintain the timestamp or access frequency of memory pages, we can use such information to prioritize the swap-out pages that should be kept in the shared memory. Clearly, this selective swap-out to disk mechanism is optimized for keeping frequently accessed pages in shared memory at the cost of maintaining the timestamp or access frequency of swap-out memory pages. To simplify the implementation and minimize the overhead of hybrid swap-out, in the first prototype of *MemFlex*, we implement the following two light weighted mechanisms for hybrid swap-out:

Most recent pages to disk. This approach is straightforward but naive. In this approach, memory pages swapped out of the VM will be written into the shared memory first. When the shared memory is full, the newly swap-out pages from the VM will be written to the VM disk swap area. The advantage of this approach is its simplicity in both design and implementation. However, the disadvantage is also obvious: most of the pages stored in the shared memory are older than the pages in the disk swap area. According to the principle of locality, programs tend to reuse data near those they have used recently. Therefore, the possibility of accessing the recent swapped out data is larger than that of fetching an 'older' swapped out page, and using the disk swap area to store the most recent swapped out pages may seriously impact the performance of a large number of applications. Also the benefit of using shared memory as the swap area will diminish as the number of pages stored on disk area is increasing.

Least recent pages to disk. This approach puts older pages to the disk swap area when the shared memory swap area is full, enabling the most recent VM swapped out pages to be kept in the shared memory. One way to organize the shared memory swap area is to use a ring buffer as shown in Figure 29. We maintain a *shm_start* pointer and a *shm_end* pointer pointing to the swapped pages with the smallest and largest offset in the buffer respectively. If a page fault comes with an offset between these two pointers, all the accesses can be served from the shared memory. Otherwise, the conventional disk based swap path will be invoked. In this design, a separate working thread is standing by and ready to be triggered to flush the pages from the shared memory to the disk swap area. In order to parallel the disk I/O and page swap operations, the working thread starts flushing the page when the shared memory is partially full, say $m\%$ full. We will evaluate how the setting of this threshold m may impact on the guest swapping and VM execution performance in Section 4.5. In the first prototype of *MemFlex*, we set the **least recent pages to disk** method as the default configuration for our hybrid swap-out model. According to [128], for the benchmark workloads such as Eclipse and SPECjbb, more than 50% of the swap-out pages stay in the swap storage on average for less than 20% of the total execution time of the workloads before being swapped in, and more than 75% of the swap-out pages stay in the swap storage for less than 28% of the total execution time. These statistics about resident time of swap-out pages are consistent with the general principle of locality and indicate that keeping the most recently swapped out pages in shared memory is more effective for hybrid swap-out than the most recent pages to disk method.

4.4.3 Proactive Swap-in

After a VM gets sufficient additional memory from the balloon driver, the VM needs to proactively swap in all the pages that are previously swapped out to the shared memory. This mechanism has two advantages. First, it reduces the number of page faults occurred in the near future. The memory paging itself is quite expensive. According to [102], it takes about an extra 1000 to 2000 CPU cycles to handle even a minor page fault, namely the required page already exists in the page cache. Otherwise, it will take even longer CPU waiting time to bring the page back into the VM memory from the secondary swap storage. The second advantage of our proactive swap-in optimization is the improved efficiency of shared memory utilization, because as soon as the proactive swap-in is completed, the corresponding shared memory swap area will be released for use by other concurrent applications.

Traditional OS provides two mechanisms to swap in pages from the disk swap area: *page faults* and *swapoff*. It is known that relying on the page faults to swap in the pages from the swap disk is expensive, and sometimes results in unacceptable delay in application performance recovery. Therefore, *MemFlex* implements the proactive swap-in by extending the *swapoff* mechanism. Concretely, *swapoff* is an OS syscall that enables to swap-in pages from the disk swap area in a batched manner. Compared with the page fault mechanism, *swapoff* allows larger amount of pages be swapped in during a shorter period of time. Once all the pages are swapped in, these pages can be accessed without any page faults. However, directly applying *swapoff* will not provide the speed-up required for fast application performance recovery for a number of reasons.

The *swapoff* syscall first checks whether there is enough free memory in the VM to hold all the currently swapped out pages, and does nothing if there is insufficient free memory. Otherwise, *swapoff* brings all the pages from the swap area to the main memory of the VM. However, this process is extremely time consuming, because the sysycall *swapoff* traverses the swap area and swaps in each page one by one. Figure 30 illustrates the process: each

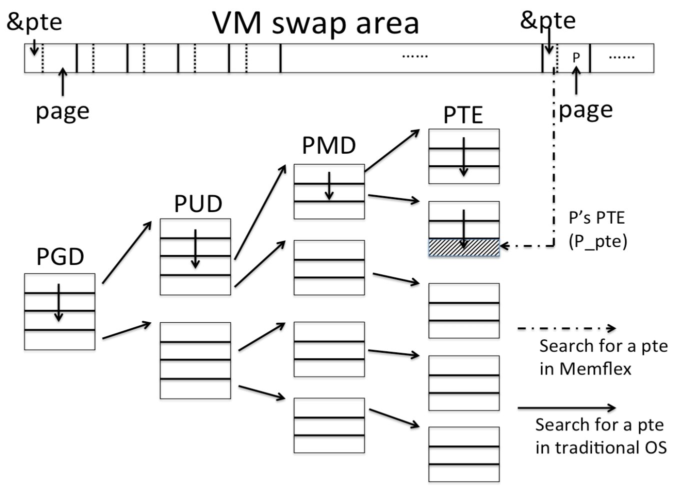

&pte
VM swap area
&pte
page
page
P
PGD
PUD
PMD
PTE
P's PTE (P_pte)
Search for a pte in Memflex
Search for a pte in traditional OS

Figure 30: Proactive swap-in v.s. Baseline swap-in.

time when a page is swapped in by the syscall *swapoff*, it needs to locate the corresponding page table entry (PTE) and update it. If a swapped out page is a shared page, then multiple PTEs need to be located and updated. Furthermore, in order to locate each PTE, *swapoff* has to scan the whole page table from the very beginning, compare each PTE with that of the page to be swapped in, until a match is found. The process of finding a corresponding PTE is annotated in Figure 30. In order to swap in the page P, which only has a single PTE (P_pte), the OS has to start from the page global directory (PGD), and traverse through all the PUD, PMD, and compare with all the PTEs until the P_pte is found.

The current design of *swapoff* is adopted in the traditional OS for two reasons: (1) The purpose of the *swapoff* syscall is to remove a specified swap device, which is assumed not to be invoked very frequently, thus a long delay can be tolerated. (2) The operation of swapping-in a page generally consists of two parts: reading the page from the swap area and updating the corresponding PTEs. The conventional swap area is a slow secondary storage, such as magnetic disk. Compared with the time spent on disk I/O wait, the cost of traversing the system wide page table for each page is relatively small. However, these reasons are no longer true in *MemFlex*. Reading a memory page from shared memory is much faster than reading a memory page from disk. Thus, compared with swapping in from the shared memory, the cost of page table scanning becomes a significantly portion of the entire cost of swapping-in a page from the shared memory swap area. Also this page table traversal cost increases as the size of the page table becomes larger.

Our idea for designing an efficient implementation of *proactive swap-in* mechanism is to maintain some meta data in the swap area, which can assist *MemFlex* to quickly locate the corresponding PTEs in the page table. Concretely, when a page is swapped out, MemFlex will keep the address of the PTEs related to this page together with the swapped-out page. Recall Figure 30, the address of the corresponding PTE is kept as the metadata in front of the swapped out page in the shared memory. For each page of 4K bytes, its meta data only takes up 4 bytes. Thus, the cost of keeping this metadata in the swap area is around only 1/1000 of the total size of the swapped out pages. For those shared pages which have multiple PTEs, we allocate a specific area in the shared memory as *PTE store*. In this case, the first byte of the meta data specifies the number of PTEs related to this page, while the lasts three bytes is an index pointing the first related PTE in the *PTE store*. When a page is swapped in, *MemFlex* is able to quickly locate the PTE(s) that needs to be updated by referring to this metadata without the need to scan the page table. The time spent on accessing the PTE of a page to be swapped in from the shared memory is only one time memory access, and it will not increase as the size of the system wide page table grows.

4.5 Evaluation

We evaluate *MemFlex* using well-known applications and benchmarks. We show that *MemFlex* improves the performance of memory intensive VMs and works well with multiple VMs running a variety of different workloads, even when the amount of shared memory for guest swapping is small and some of the swapped out pages have to be routed to disk swap partitions. We present our experimental evaluation results to answer the following set of questions:

- By using the same amount of memory, how much application performance improvement can *MemFlex* bring compared with directly allocating the shared memory to the VMs beforehand? (Section 4.5.1)

- How do the applications perform when the shared memory swap space reserved by *MemFlex* is not big enough to hold all the swap traffic from the VMs? (Section 4.5.2)

- When the VM has successfully reclaimed sufficient free memory, how fast can *MemFlex* help the VM to swap-in its pages resident in the host coordinated shared memory back to the VM memory? (Section 4.5.3)

- How will the performance of *MemFlex* be affected by the settings of the system parameters, such as the value of *SWAPFILE_CLUSTER*? (Section 4.5.4)

- How does *MemFlex* compare with other existing systems in terms of guest swapping performance? (Section 4.5.5)

- Can *MemFlex* scale in large scale virtualization platforms? (Section 4.5.6)

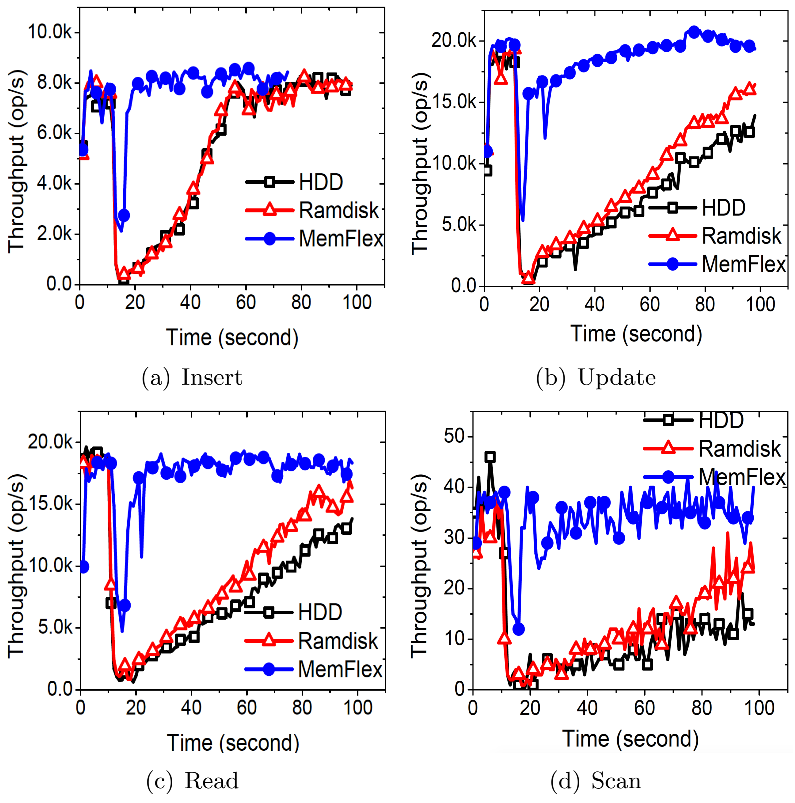
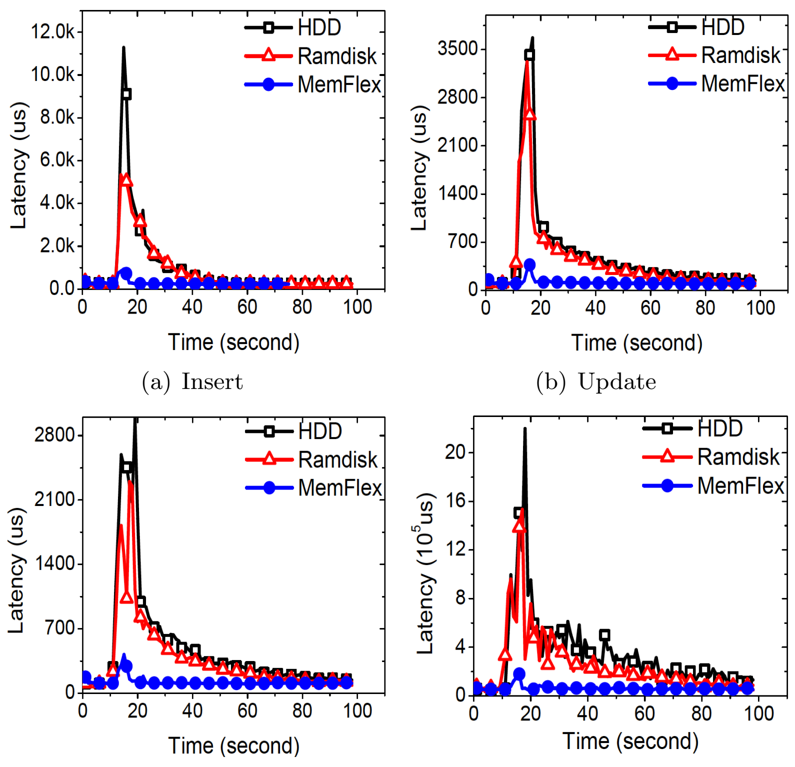


Figure 31: Throughput of Redis server measured by YCSB workloads

Experiments Setup. Most of our experiments are conducted on an Intel Xeon based server provisioned from a SoftLayer cloud [24] with two 6-core Intel Xeon-Westmere X5675 processors, 24GB DDR3 physical memory available for the guest VMs, 1.5 TB SCSI hard disk, and 1Gbit Ethernet interface. The host machine runs Ubuntu 14.04 with kernel



(a) Insert (b) Update

(c) Read (d) Scan

Figure 32: Latency of Redis server measured by YCSB workloads

version 4.1.0, and uses KVM 1.2.0 with QEMU 2.0.0 as the virtualization platform. The guest VMs also run Ubuntu 14.04 with kernel version 4.1.0. 4 VMs are simultaneously running on the host. We use a memory intensive application **memAlloc** to allocate and scan a large amount of memory in the VM in order to simulate high VM memory pressure of varying intensity. The memory utilizations of VMs are monitored at an interval of 5 seconds to trigger the balloon driver when necessary. We also evaluate *MemFlex* in a larger scale virtualization setup, in which 8VMs are deployed on a physical machine with 64GB available memory for guest VMs. We compare the following cases in most of the experiments.

- **HDD case:** Each VM is assigned with up to 6GB memory.
- **MemFlex case:** Each VM is assigned with up to 5.5GB memory, while the remaining 2GB memory is reserved for the host as the available shared memory area.
- **Ramdisk case:** Similar to the *MemFlex* case, except that instead of using shared memory, the available 2GB memory is organized as a ramdisk, which is mounted to

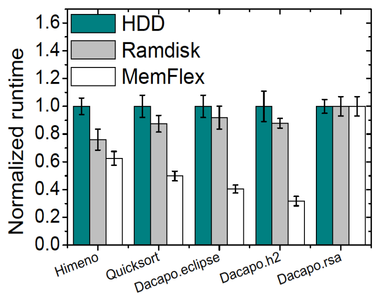


Figure 33: Normalized runtime

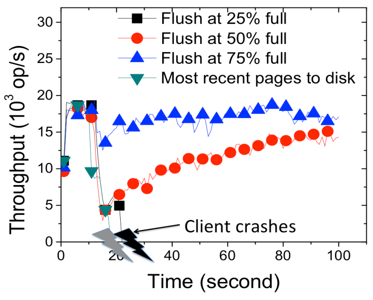


Figure 34: Hybrid swap-out (Read)

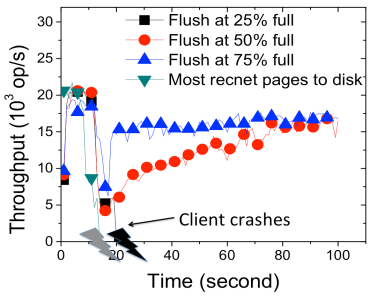


Figure 35: Hybrid swap-out (Write)

the VMs.

The following benchmarks and applications are used throughout the experiments:

Redis [22]. An open source, in-memory key-value store, which supports a wide spectrum of data structures such as strings, hashes, lists, bitmaps, and is often used as database, cache, and message broker.

Dacapo [4, 43]. A benchmark suite consisting of a set of open source, real world Java applications with non-trivial memory loads. Example applications are *h2*, a JDBCbench-like in-memory benchmark for executing a number of transactions against banking application; *eclipse*, which executes some of the (non-gui) jdt performance tests for the Eclipse IDE, and *xalan*, which transforms XML documents into HTML.

SPECJVM2008 [25]. A set of real applications such as the *javac compiler* as well as area-focused benchmarks, such as *xml, crypto* with the focus on the performance of the JRE executing a single application.

Himeno [8]. It is developed to evaluate the performance of incompressible fluid analysis code. This benchmark takes measurements for solving the Poisson equations using the Jacobi iteration method. The performance of Himeno is especially affected by the performance of memory subsystems.

Sysbench [27]. A benchmark suite that allows users to evaluate the system performance by manually creating workloads for a specific subsystem. For example, users can benchmark their CPU by generating CPU intensive workloads from Sysbench. We use this benchmark to generate disk I/O intensive workloads and evaluate how they interfere with the VM memory swapping operations.

QuickSort. This application is written in C by ourselves. We feed it with large data sets to make it memory intensive.

4.5.1 Overall Performance of MemFlex

In this section, we use Redis and a number of benchmark applications (e.g., Himeno, Decapo, Quicksort) to evaluate the effectiveness of MemFlex.

Redis. We evaluate how *MemFlex* can improve the performance of the Redis. A Redis server is running on VM1, which is pre-loaded with 5GB data in memory, while the other 3 VMs are idle. Different workloads generated by YCSB [53] are executed in a remote machine, which is connected to VM1 as a client. We let the client run for about 10 second, and then start *memAlloc* in the VM where the Redis server is running. The *memAlloc* will allocate and initialize 7GB memory to increase the total memory demand of VM1 to 12GB. In the *HDD* case, the balloon driver will move 2GB memory from each of the remaining three idle VMs (VM2, VM3 and VM4) to VM1, while in the *MemFlex* case and the *Ramdisk* case, the balloon driver will move a total of 4.5GB to VM1 with 2GB memory from VM2 and 2.5GB from VM3, while the reserved shared memory is used by VM1 as its swap area. We measure the throughput as well as the latency of the Redis server, and compare the Redis server performance among the three cases. The workloads used in this set of experiments include *Insert, Read, Update, and Scan*, and are generated by YCSB with a uniform request distribution.

Figure 43 displays the results of how the Redis server performs in terms of throughput. We make several interesting observations. First, the Redis throughput drops significantly at around the 10th second in all the cases, no matter which workload is running. This is due to the execution of *memAlloc*. Since there is not enough free memory in VM1 when *memAlloc* starts. However, VM1 cannot get free memory from the balloon driver immediately, which causes VM1 to swap out some of its memory pages, causing serious degradation of the Redis performance. We also observe that the performance in *MemFlex* also drops at the 10th second, this is because *memAlloc* also consumes CPU cycles to allocate the large amount of memory when it starts to run, which results in severe CPU competition within VM1.

Second, in both the *HDD* case and the *Ramdisk* case, the performance of all workloads can recover from the performance drop, but very slowly. The slow performance recovery indicates that after VM1 gets enough free memory from the balloon driver, the swap-in process becomes the dominating bottleneck. When VM1 needs to access some page that has previously been swapped out, a page fault will be triggered and the page needs to be read back from the swap device to the memory of VM1 through disk I/O. Both page fault and disk I/O are very expensive operations. In comparison, for the *MemFlex* case, the Redis performance recovers significantly faster. For the *Read* workload in Figure 43(c), its throughput drops from 17813 OP/sec to 7049 OP/sec at the 10th second, but it only takes about 5 seconds for the throughput to be fully recovered in *MemFlex* whereas it will take more than 80 seconds for the *Ramdisk case* and much longer for the *HDD case* to fully recover to the throughput performance prior to the launch of *memAlloc*.

It is worth to note that although the swap device *Ramdisk* is also using the host memory for swap, its performance, though slightly higher than that of the *HDD* case, is substantially lower than *MemFlex*. This experimentally proves our analysis in Section 4.2: when the swap area is mounted to the VM as a block device, the overhead of block I/O processing and the context switch between the VM and the host is dominating the efficiency of VM swapping. Thus simply changing the VM swap device from disk to ramdisk in host memory will not guarantee sufficient VM performance improvement. This observation further validates the superiority and originality of the *MemFlex* shared memory swapper.

Figure 32 displays the latency measured by the *Insert, Read, Update and Scan* workloads. Similar to the observations from Figure 43, the execution of *memAlloc* at around 10th second introduces significant performance degradation to the Redis server, which causes the sudden latency increase at that time. As the balloon driver inflating more memory at 5 seconds interval, the performance of Redis server in both *HDD* case and *Ramdisk* case can recover, but much slower than that of the *MemFlex* case. Although we can see some tiny

Table 5: Time (nano seconds) spent on *Page read* and *PTE update* when swapping in 2GB data. "T" means "Total time."

	Overall	Per page			
	T(sec)	T(ns)	Page read(ns)	PTE update(ns)	Others(ns)
HDD	212	473145	17358 (24%)	354561 (75%)	1226 (1%)
MemFlex w/o opt	151	337611	3305(1%)	333172 (98%)	1131(1%)
MemFlex w/ opt	4	4081	3288(18%)	284 (7%)	509(13%)

bump of the latency of Redis server in the *MemFlex* case, due to the execution of *memAlloc*, it recovers very quickly, in just a few seconds, thanks to the *MemFlex* shared memory swapper.

Other applications. We also measure the effectiveness of MemFlex for other benchmark applications, such as Himeno, Dacapo.eclipse, Dacapo.h2, and Crypto.rsa. The setup is similar to the Redis experiments. At the beginning, five applications - *Himeno, Quick, Dacapo.eclipse, Dacapo.h2* and *Dacapo.rsa* are running simultaneously in VM1, which takes a total of around 5 GB memory. Then, *memAlloc* starts running at the 10th second, and it brings the total memory demand of VM1 to 12GB. Figure 33 compares the normalized execution time of each application under the three cases. For *Himeno, Quicksort, Dacapo.eclipse* and *Dacapo.h2*, *MemFlex* provides the shortest execution time, while the *HDD case* has the longest in execution time. Also the execution time of *crypto.rsa* remains almost the same under all three cases, because unlike the other four memory intensive workloads, *crypto.rsa* is a CPU intensive application. This also demonstrates that *MemFlex* is light weighted and does not impact the performance of CPU intensive applications running in the VM.

4.5.2 Hybrid Swap-out

In this section, we evaluate the effectiveness of the hybrid swap-out mechanism in *MemFlex* by considering the cases when the shared memory space is not enough to hold all the pages swapped out from the VM.

We evaluate the performance of a Redis server VM in this set of experiments. The setup is similar to that in Section 4.5.1, the total memory demand is 12GB after *memAlloc* starts running at about the 10th second. However, instead of 12GB, only 11GB is finally given to

VM1 (10GB as main memory and 1GB as the swap area). The other 1GB will be swapped out to disk. Both the *ReadIntensive* and *WriteIntensive* workloads are generated by YCSB, and executed in a remote client machine.

Figure 34 and Figure 35 display the Redis server performance. We make a number of observations. First, for both workloads, the client crashes if the *most recent pages to disk* mechanism is used. This is because when the client tries to access a recent swapped out page on disk, the latency exceeds the timeout set by the Redis client. Second, the hybrid swap-out using the *least recent pages to disk* strategy performs better than the *most recent pages to disk* strategy, while setting the flushing threshold at 75% performing the best followed by 50%. When the threshold value is set to 25%, which means that *MemFlex* starts to flush pages from the shared memory to the disk when the shared memory is only 25% full, then the client will still crash, just a few seconds later than the case of using the *most recent pages to disk* strategy. This is because a smaller threshold value will make *MemFlex* to flush the pages from the shared memory to the disk unnecessarily earlier, which also leads to low utilization of the shared memory swap area.

4.5.3 Proactive Swap-in

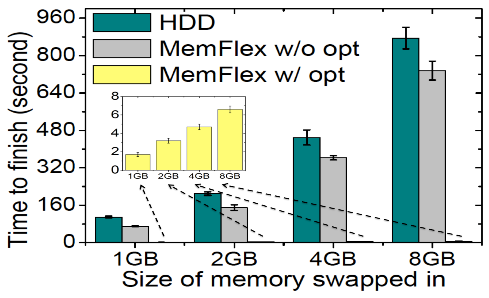
In this section, we evaluate the effectiveness of the *proactive swap-in* mechanism in *MemFlex*. The purpose of *proactive swap-in* is to let the VM quickly swap in those previously swapped out memory pages resident in the shared memory as soon as the VM gets sufficient free memory. This has two obvious advantages: it reduces the number of expensive page fault operations in memory accesses and it quickly frees up the shared memory swap area for other applications.

We start a Redis server on VM1 with 6GB main memory and 2GB swap area. A client then loads 6GB data to the Redis server VM to fully fill the memory. Then we start *memAlloc* to demand 2GB memory from the VM. This will trigger 2GB of the Redis data to be swapped out to the VM1 swap area. Finally, we terminate *memAlloc* so that the 2GB memory in VM1 will be released and *MemFlex* will proactively swap in the 2GB swapped out memory pages from the shared memory swap area back to VM1. Various swap related statistics are measured.

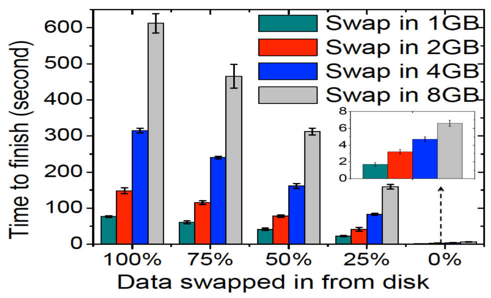
Recall Section 4.4.3, the process of swapping-in a page consists of three parts: *Page read, PTE update* and *others*. Table 5 shows how much time is spent on each part in order to swap-in a page from the swap area under three cases: (1) *HDD swap-in*, (2) *MemFlex without (w/o) proactive swap-in optimization (MemFlex w/o opt)*, and (3) *MemFlex with (w/) proactive swap-in optimization (MemFlex w/ opt).*

We make three observations from Table 5. First, in the *HDD swap-in* case, the time spent on *Read* and *PTE Update* occupies 25% and 74% of the total swap-in time. This is partly because disk I/O is very slow and partly because in order to swap-in a single page, the OS needs to scan the page table to find the corresponding page table entries that it needs to update, which is very time consuming, and this cost will increase as the size of the page table increases. Second, compared to the *HDD swap-in* case, the average page swap-in time in the *MemFlex w/o opt* case is improved by 28.6% from 473145 nano seconds (ns) to 37753 ns. By zooming into the time spent on each part, we find that this improvement is mostly due to the fast *Read* speed, because the time spent on *PTE Update* is now occupying 98% of the total time. This also validates our analysis in Section 4.4.3 that when using the shared memory for VM swap traffic instead of disk swap, the *PTE Update* becomes the major bottleneck for swap-in events. With the help of *proactive swap-in*, the *PTE Update* time is significantly decreased from 333172 ns to 284 ns. Because in this case, the OS swaps out not only memory pages but also some metadata, which can help the OS to quickly identify which page table entries need to be updated during the swap-in process. Thus the cost of scanning the page table is avoided.

Next, we measure the total time spent on swapping-in different amount of memory pages under different scenarios. Figure 36(a) shows that in both *HDD swap-in* case and *MemFlex w/o opt* case, the total time spent on swap-in will increase dramatically with the increase of the total amount of memory that needs to be swapped in. For example, in the *HDD swap-in* case, it takes 110 seconds to swap in 1GB memory and 875 seconds to swap in 8GB memory. However, with *proactive swap-in*, the total time consumed by the same swap-in events is two orders of magnitude shorter than the *HDD swap-in* case and the *MemFlex*



(a) Swap in from shared memory



(b) Hybrid Swap in

Figure 36: Total time spent on swapping in

Table 6: Number of VM page faults from each application

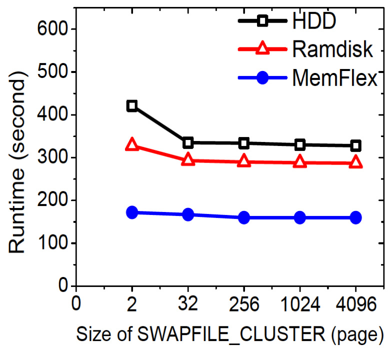
	QuickSort	Himeno	eclipse	h2
MemFlex w/o opt	46K	109K	38K	45K
MemFlex w/ opt	6K	8K	14K	31K

w/o opt case. Also as the size of the memory to be swapped-in grows, the time spent on swap-in also grows though slowly. For instance, in the *MemFlex w/ opt* case, swapping in 1GB memory needs 3 seconds, while it takes only 6 seconds to swap in 8GB memory. Figure 36(b) shows the swap-in performance when the swap area is a mix of shared memory and disk. Compared with the *HDD swap-in*, even if all the data is swapped in from the disk swap area, *MemFlex* saves 30% swap-in time compared to the *HDD swap-in* in Figure 36(a). Also, as the portion of on-disk data decreases, the total swap-in time also decreases.

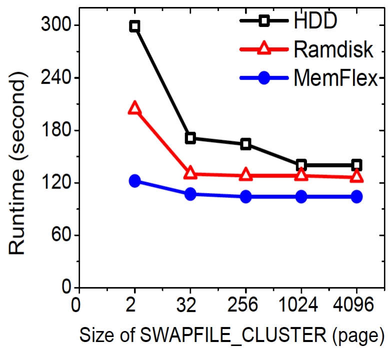
Finally, we evaluate how *proactive swap-in* help reduce the number of page faults. We run four applications (QuickSort, Himeno, Eclipse, h2) sequentially on VM1 which has insufficient memory, thus the applications start to experiencing guest swapping. Then we use the balloon driver to give the VM more than enough memory. The page faults of each application are recorded. Table 6 shows that *MemFlex proactive swap-in* can reduce the number of page faults by 33% to 90%.

4.5.4 Effect of Swap Area Allocation

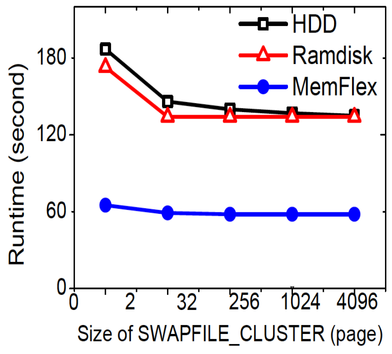
In this section, we evaluate how the settings of the system parameter *SWAPFILE_CLUSTER* may impact on the effectiveness of *MemFlex*. The Linux kernel allocates the swap area in the unit of *SWAPFILE_CLUSTER* pages. Thus, a larger value of *SWAPFILE_CLUSTER*



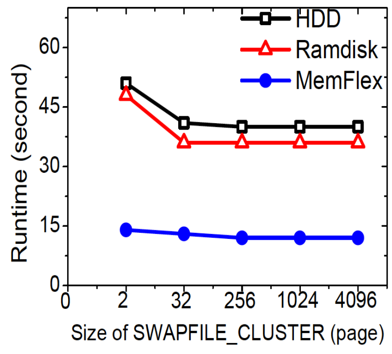
(a) Quicksort



(b) Himeno



(c) Dacapo.eclipse



(d) Dacapo.h2

Figure 37: Effect of the system parameter SWAPFILE_CLUSTER on *MemFlex*

reduces the fragments of the swap area, and leads to more sequential I/O than random I/O. While a smaller value of *SWAPFILE_CLUSTER* makes the swap area allocation more flexible.

In this set of experiments, we use four benchmark applications: *Quicksort, Himeno, Dacapo.eclipse* and *Dacapo.h2*. All applications are executed on VM1, and the memory swap happens during the execution of each application. We vary the value of *SWAPFILE_CLUSTER* from 2 to 4096, and measure the execution time of each application by setting different values of *SWAPFILE_CLUSTER*.

Figure 37 compares the runtime of the applications under three scenarios: *HDD, Ramdisk, and MemFlex*. First, no matter in which scenario, the larger the value of *SWAPFILE_CLUSTER* is, the smaller the execution time of the application is, though the decrease rate is sublinear. For example, in HDD swap scenario, the execution time of *Himeno* decreases from 299 seconds to 171 seconds and 164 seconds when the value of *SWAPFILE_CLUSTER* increases from 2 to 32 and 256 respectively. This is because a larger value leads to more sequential

I/O than random I/O and the performance of sequential I/O is much better than that of random I/O for disk. Second, for all the applications running with *MemFlex*, the value of *SWAPFILE_CLUSTER* has very little impact on their runtime. This is primarily because *MemFlex* uses shared memory as the VM swap area, and there is very little difference between the performance of random access and sequential access. Consider *Dacapo.eclipse*, its runtime only slightly decreases when the value of *SWAPFILE_CLUSTER* increases from 2 to 256 and when the value of *SWAPFILE_CLUSTER* further increases from 256 to 4096, its runtime stays almost the same, as shown Figure 37(c). The reason of the slightly runtime decrease when *SWAPFILE_CLUSTER* increases from 2 to 256 is due to less frequent invocation of swap memory allocation in the kernel. Third, no matter what specific value is used to set *SWAPFILE_CLUSTER*, *MemFlex* is able to provide the best performance for all the applications, *Ramdisk* Swap comes to the second, and *HDD* swap is the worst.

4.5.5 Comparison to Other Swap Systems

In this section we compare *MemFlex* to VSwapper [39], a collaborative memory swapper for virtualized environment. It enhances the baseline swapping performance by avoiding various types of superfluous/inconsistent swap operations. We conducted the same set of experiments with the *Dacapo.eclipse* workload as in [39], and compare its performance with *MemFlex*. Figure 38 shows the performance of *MemFlex* compared with that of *Vswapper* and the *Baseline*, in which the VM swaps to the disk. While the performance of the *Baseline* case is the worst, the execution time of *Dacapo.eclipse* in *MemFlex* is up to 3x faster than that in *VSwapper*. It is worth noting that the experimental evaluation of VSwapper reported in [39] uses very small VM memory sizes ranging from 512 MB to 256 MB. To be fair with the best case scenarios reported in [39], we keep the comparison with the same VM memory sizes.

4.5.6 Larger Scale Experiments

This set of experiments evaluates *MemFlex* in a larger scale virtualization setup by deploying 8VMs on a physical machine with 64GB free memory. For the case *without MemFlex*, each VM is assigned with 8GB memory, and loaded with 6GB Redis data. At the same time, a

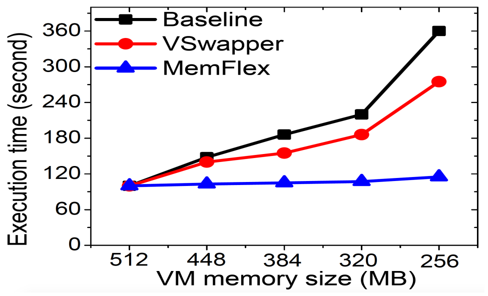


Figure 38: System comparison

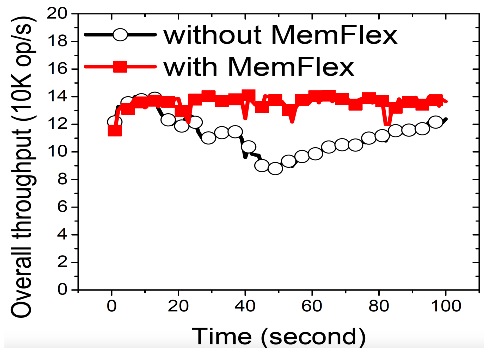


Figure 39: Redis workloads

remote client is connecting to each of these 8 VMs and executing a read intensive workload. Four *memAlloc* applications are randomly and sequentially executed on four of the eight VMs and each of them requests for 4GB memory. The settings for the case *with MemFlex* are similar with the only difference that in this case, each VM is assigned with 7GB memory while the remaining 1GB per VM (a total of 8GB) is reserved at the host for *MemFlex*. Also, the VM memory utilization are monitored at an interval of 5 seconds and the memory are moved among VMs via the balloon driver when necessary.

All the client workloads start at the same time and lasted for 100 seconds. Figure 39 compares the total Redis throughput for the 8 VMs with *MemFlex* and without. We find that *without MemFlex*, although some degree of performance recovery can be observed, the overall trend of throughput is declining until reaching the 46th second. During this period of time, the occasionally recover is due to the fact that the balloon driver is moving memory to the specific VM. In the *MemFlex* case, we can clearly observe that the total throughput has not been affected much by the four random execution of *memAlloc*. Even though there are small performance degradations, which can recover fairly quickly. This set of experiments shows that with unpredictable memory demands in a larger scale virtualization platform, *MemFlex* is able to offset the delays of the balloon driver by leveraging shared memory swap.

4.6 Conclusions

We have presented the design of MemFlex, a highly efficient shared memory swapper. This chapter makes three original contributions. First, *MemFlex* can effectively utilize host idle memory by redirecting the VM swapping traffic to the host-guest shared memory swap area. Second, *MemFlex* hybrid memory swapping model promotes to use the fast shared memory swap partition as the primary swap area whenever possible, and smoothly transits to the conventional disk-based VM swapping scheme on demand. Third but not the least, *MemFlex* proactive swap-in optimization offers just-in-time performance recovery by replacing costly page faults with an efficient swap-in implementation. We evaluate *MemFlex* using a set of well-known applications and benchmarks and show that *MemFlex* offers up to two orders of magnitude performance improvements over existing memory swapping methods. We have implemented the first prototype of *MemFlex* on the KVM platform and we are currently working on deploying *MemFlex* on the Xen platform by utilizing the relevant kernel functions and kernel data structures provided in Xen hypervisor.

Chapter V

MEMLEGO - IMPROVING MEMORY EFFICIENCY FOR HIGH PERFORMANCE VIRTUAL MACHINE EXECUTION

Memory is increasingly becoming a bottleneck in virtualized systems. Mechanisms for utilizing memory efficiently are widely recognized as critical system optimizations for high-performance virtual machine (VM) execution. Memory overcommitment and memory ballooning are the most popular systems level mechanisms for improving memory utilization. However, these mechanisms often relies on accurate estimation of VM working set size at runtime, which is difficult under changing workloads. This chapter presents MemLego, a shared memory based memory optimization framework for managing and improving memory efficiency in virtualized environment. With MemLego, each VM starts with an application-specified lower bound of memory. MemLego maintains a shared memory region across multiple VMs, and enables those VMs under memory pressure to obtain additional memory on demand. MemLego makes three original contributions: It offers on-demand VM memory allocation and deallocation in the presence of changing workloads. It relieves VM execution performance from drastic degradation due to memory swapping. It provides shared memory pipes for high-performance communication between co-resident VMs. Extensive experiment results show that MemLego offers up to 4 times throughput enhancement for Memcached and Redis, up to 2 orders of magnitude performance improvements over conventional memory swapping methods, and improves the throughput of native inter VM communication by up to 45 times.

5.1 Introduction

Main memory is a critical and shared resource in virtualized computing environment. As the number of CPU cores doubles approximately every 2 years, the DRAM capacity is doubling roughly every 3 years [90] and as a result, the memory capacity per core is expected to drop by about 30% every two years [1]. The trend is worse for memory bandwidth per core [101]. At the same time, with the upsurge of big data processing and big data analytics, main memory is increasingly becoming a bottleneck in virtualized systems. Large portions of big data software and application code are written today to maximize the use of memory and minimize access to high latency storage. For example, in memory key-value stores, such as Redis [22] and Memcached [12], are popular in-memory stores for big data workloads. Spark [126], Flume [76], Kafka [69] are popular big data computing platforms for memory-intensive applications. However, memory utilization is not satisfying and improving memory efficiency is becoming an important optimization goal for many big data systems and applications. Recent statistics from Google datacenter traces [50] show that although more than 90% of the memory has been allocated, only less than 50% of them has been used. One of the main reasons is that the memory allocated to each virtual machine (VM) has to satisfy its peak demand, thus idle memory exists most of the time.

Advances in computer hardware technologies tackle the memory bottleneck problem along two dimensions. (1) Various DRAM optimization technologies have been proposed and developed for improving DRAM parallelism [45, 82], latency and energy [58, 59, 62, 88, 87], and minimizing memory capacity and bandwidth waste [105, 104, 103]. (2) Emerging memory technologies, such as non-volatile random-access memory (NVRAM) technologies, and hybrid main memory systems, are proposed and being deployed. Research and development (R&D) efforts along both dimensions confirm the importance of achieving efficient memory utilization under changing workloads. [101] provides a good review of problems and challenges in memory systems from computer hardware research perspective.

Orthogonal to the advances in memory hardware technologies, research efforts for improving memory efficiency have also been engaged from software design and software optimization perspective by researchers in systems, networking, programming language, compiler and database systems. The most relevant research efforts to MemLego are centered on developing memory resource partitioning and prioritization mechanisms to achieve high performance for all applications, such as memory balancing research in virtualized systems. Although virtualization allows multiple virtual machines (VMs) to run simultaneously on a single hardware platform by demand-driven CPU time slicing [38, 107, 91], it remains technically challenging to share main memory among VMs in a timely fashion by demand-driven memory resource partitioning and prioritization. Memory overcommitment and balloon driver technology are the state of art software technology for moving memory from one VM to another. However, the accurate detection of the right timing to turn on the balloon driver is known to be a hard problem, which involves decision on *when* a VM needs more memory, *how much* it needs and from *where* it can get the free memory. As a result, most of the systems administrators today pre-configure the balloon driver at a fixed ballooning interval (e.g., 30 seconds [120]) with a fixed amount of ballooning memory. Such static configuration of memory partitioning and prioritization could severely degrade VM performance under changing workloads. Recently, several dynamic VM memory balancing approaches [75, 80, 134] have been proposed. Most of them are based on estimation and prediction of VM memory working set at runtime. However, accurate VM memory estimation is not only difficult [39], especially under changing conditions, but can also incur high performance overhead [133] due to frequent VM memory access interception. Thus, the trade-off between balancing frequency and accuracy is rest on the shoulders of systems administrators in practice.

With these problems in mind, we present the design and implementation of MemLego, a shared memory based optimization framework for improving memory efficiency and achieving high performance virtual machine execution. MemLego optimizes VM memory efficiency from a number of dimensions. First, MemLego reserves and manages a shared memory region at the host for sharing among all hosted VMs and provides a light-weight mechanism for on-demand VM memory allocation and de-allocation that are proportional to their workloads requirements. Second, to effectively utilize host idle memory, MemLego also promotes a shared memory based optimizations for fast memory swapping by providing a hybrid swap-out and fast swap-in facility. The third optimization that MemLego advocates is an inter-VM shared memory pipe for improving inter-VM communication efficiency. We evaluate MemLego through extensive experiments with different benchmarks on a set of representative application workloads. Our experiment results show that MemLego offers up to 4 times throughput enhancement for Memcached and Redis, up to 2 orders of magnitude performance improvements over conventional memory swapping methods, and improves the throughput of native inter VM communication by up to 45 times.

5.2 Related Work

The most relevant existing research efforts are dynamic memory balancing by host-guest coordination and shared memory based computing and communication optimizations.

Dynamic memory balancing. We categorize this line of research into three threads. The first thread is centered on redesigning operating system (OS) to enable more efficient host-guest coordination. The transcendent memory (tmem) on Linux by Oracle and the active memory sharing on AIX by IBM PowerVM are the two representative efforts. The transcendent memory [96] allows the VM to directly access a free memory pool in the host, which can be used by Guest OS to invoke the host OS services and by the host OS to obtain the memory usage information of the guest VM [85]. For the applications that implement their own memory resource management, such as database engines and Java virtual machines (JVMs), [112] proposes to use application-level ballooning mechanisms to reclaim and free memory. However, most of the proposals in this thread rely on some serious changes to guest OS or applications, making the solutions harder for wide deployment.

The second thread centers around using host coordinated ballooning for VM memory overcommitment. The main idea is to embed a driver module into the guest OS to reclaim or recharge VM memory via the host. The balloon driver, proposed in 2002 [120], has been widely adopted in mainstream virtualization platforms, such as VMware [109], KVM [84], Xen [41]. A fair amount of research has been devoted to periodic estimation of VM working set size because an accurate estimation is essential for dynamic memory balancing using the balloon driver. For example, VMware introduced statistical sampling to estimate the active memory of VMs [120, 29]. Alternatively, [133] builds and updates the page-level LRU histograms by having the hypervisor intercepting memory accesses from each VM and uses the LRU-based miss ratio to estimate VM memory working set sizes. [134] proposed to implement the page-level miss ratio estimation using specific hardware to lower the cost of tracking the VM memory access. However, several independent research efforts [80, 94, 75] show that accurate VM working set size prediction is difficult under chaining conditions.

The third thread includes complimentary techniques to improve dynamic memory consolidation, ranging from memory hotplug, collaborative memory management to remote memory swapping. Memory Hotplug [92, 113] was proposed to address the problem of insufficient memory or memory failing at both guest VMs and host. It refers to the ability to plug and unplug physical memory from a machine [92] without reboot to avoid downtime. Collaborative memory management [114] proposed a memory access monitoring based information sharing mechanism between host and guests to reduce the host paging rate and improves the throughput of memory allocation requests for a guest. Vswapper [39] tracked correspondences between disk blocks and guest memory pages to avoid unnecessary disk I/O caused by uncooperative memory swapping between guest and host. However, Vswapper does not provide any mechanisms to utilize host idle memory. In addition, several efforts have engaged in combining swapping to the local disk with swapping to the remote memory via network I/Os [63, 115, 123, 128].

Shared memory based optimization. The other relevant research efforts are shared memory based performance optimization. The most representative work is distributed shared memory management (DSM) [40, 66, 79, 83], which can benefit the applications by providing them a globally shared virtual memory even though they execute on separated physical nodes. Shared memory has also used in multiprocessor systems for performance acceleration and programming abstraction. Tornado [68] improved locality and concurrency of applications by designing an optimized shared memory multiprocessor operating system. vNUMA [46] is designed to enable legacy applications and operating systems to run on a cluster by abstraction of a cluster as a virtual shared memory multiprocessor. [125] proposed a shared memory processor design that can improve the correctness of multi-thread programs.

To the best of our knowledge, MemLego is the first effort that explores the efficient use of host idle memory as shared memory resource to provide light weight and on demand memory allocation, fast and hybrid memory swapping, and shared memory pipe for optimizing co-resident inter-VM communication.

5.3 System Design

In this section, we describe the design of MemLego core system components: *ShmManager* and *MemExpand*. The former is responsible for establishing a shared memory channel between the host and the VMs to enable flexible and on demand sharing of the free memory at the host. The latter is responsible for providing dynamic allocation and de-allocation of shared memory based on the workload demands of respective VMs. Figure 40 shows a sketch of the MemLego system architecture. We will illustrate the detail in the subsequent sections.

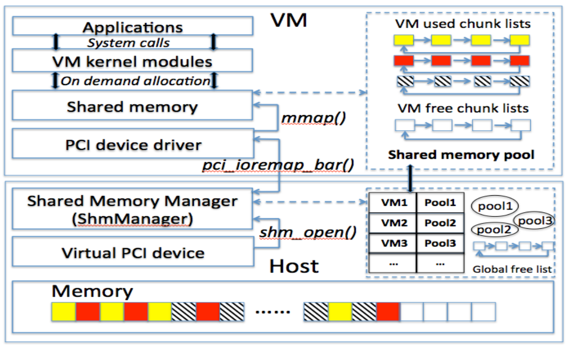


Figure 40: MemLego Architecture

5.3.1 Establishing Shared Memory Channel

MemLego is implemented as a shared memory optimization layer between the host kernel and the VMs running on the host. The main job of the *ShmManager* is to enable multiple VMs to access the host-guest shared memory region in a coordinated manner. Before launching VMs on the host, the MemLego Initiator allocates and initializes a segment of the host free memory as the reserved shared memory region managed by MemLego. It also creates an emulated virtual PCI device as the bridge between the host and the VMs such that a virtual PCI device can be mounted to each VM. A PCI device has several base address registers (BARs), which is used to specify the virtual memory regions that this PCI device can use. Therefore, the shared memory allocated by the Initiator is assigned to one of the BARs of the PCI device as its memory region. Inside a VM, a PCI device driver is created, which maps the PCI device memory into its kernel space. The device driver will be invoked once the driver in the VM is executed. For example, *pci_ioremap_bar(pci_dev, 2)* requests the kernel to map the memory region specified by BAR2 into the VM's kernel address space. After the VM kernel maps a memory region into its kernel address space, the other kernel modules as well as the user-level applications can access the shared memory.

The two interface functions that ShmManager provides to the VMs are *unsigned long shm_malloc(size_t size)* and *void shm_free(unsigned long offset, unsigned long len)*. The former allocates a piece of shared memory from the shared memory region with the capacity specified by the input parameter *size*, and returns to the VM the offset in the shared memory region where the allocated shared memory piece starts. The *shm_malloc()* returns -1 when there is insufficient free shared memory to satisfy the allocation request. The latter API function allows the VM to free a pre-allocated piece of shared memory.

5.3.2 Organizing Shared Memory

In order to support flexible shared memory management, the shared memory region in the host is divided into shared memory chunks of fixed size (e.g., 4KB per chunk) and free chunks are maintained using linked list. This enables the shared memory to be allocated to or revoked from the VMs chunk by chunk. In MemLego, we organize the shared memory chunks into three types of linked lists: *active, inactive*, and *idle. Active chunks* refer to the ones that have allocated to some VM and currently contain valid data. *Inactive chunks* are those that have been allocated to some VM but have not been or no longer be actively used by the applications running on the VM. When a chunk does not belong to any VM, this chunk is an *idle* chunk. Chunks may dynamically convert from one type to another based on their current usage. For each VM, we maintain an *active_chunks* list and an *inactive_chunks* list. In addition, all the free chunks that have not been allocated to any VM will be put together using a linked list, named *free_chunks*.

Three types of metadata, *free_shm, list_descriptor,* and *chunk_descriptor*, are maintained to facilitate the coordinated access to the shared memory pool by multiple VMs. *free_shm* is a global meta data which locates in the very front of the shared memory region. This metadata indicates the offset of the first element in the *free_chunks*. Whenever MemLego wants to allocate a shared memory chunk to a VM, it needs to check this metadata first. Given that this is a global metadata that is concurrently accessed and updated by multiple VMs, in order to guarantee the update consistency, a global lock is assigned to *free_shm*, and every VM needs to successfully acquire this lock before reading and updating the *free_shm*. The *list_descriptor* contains the information about a single linked list that belongs to some VM. For example, it may maintain two pointers pointing to the start chunk and the end chunk that store the valid data, so that when new data arrives, it can be appended next to the end of the current valid data. Since a chunk may be partially used, each *chunk_descriptor* records which part of the data in the chunk is valid, while the rest of chunk is free.

5.3.3 On Demand Memory Allocation

The second core component of MemLego is *MemExpand*, which is built on top of ShmManager to provide on demand memory allocation from the host shared memory to a VM that is under memory pressure. The two basic interface functions of MemExpand is *shm_malloc()* and *shm_revoke()*. When a VM needs more memory resource, it can use the *shm_malloc()* to allocate some free shared memory chunks from the shared memory pool maintained by MemLego, for example, by removing the newly allocated chunks from the *free_list* and adding them to the corresponding *active_chunk* linked list for the respective VM. Similarly, a separate thread is maintained, which periodically checks the utilization of the linked list in each VM, and removes the *inactive* chunks when possible. In order to prevent information leakage, when removing memory chunks from the linked list in a VM, all data content is fully erased. This guarantees that each newly allocated shared memory chunk is completely empty, and a VM will not leak any data to the other VMs.

The implementation of the *shm_malloc()* needs to address two issues: (1) how to allocate the shared memory to a VM on demand, and (2) how to detect when a VM is under memory pressure and thus turn on shared memory allocation. To address both issues, we need to examine how applications running on a VM request memory. Generally speaking, applications request memory from the operating system by either declaring an array or use the library call *malloc()*. Declaring an array helps the applications to get memory from its stack space, and the lifetime of this memory region is only within the function in which the array declaration is called. In other words, the allocated array will be automatically destroyed when the function finishes. At the same time, the applications can get memory from its heap space by using *malloc()*, and the memory region allocated by *malloc()* can last until a *free()* has been explicitly invoked. Although allocating memory from the stack is easier and faster, allocating from the heap space using *malloc()* is a more flexible approach for large memory allocation and it is also widely used by a large number of applications for their memory allocation for a number of reasons: (1) Since the size of heap space is much larger than that of the stack space, large block of memory is usually allocated from the heap space by using *malloc()*. (2) The duration of the memory allocated from the heap can be explicitly controlled by the application. (3) The size of the memory regions allocated by *malloc()* can be dynamically changed, while the array based allocation from stack is static and fixed.

In the first prototype implementation of MemLego, the MemExpand intercepts the library call *malloc()* in the guest and replaced it with its own function named *expand_malloc()*. This approach is light weight and transparent to both the applications and the VM kernel, and thus neither applications nor guest OS kernel need to be modified. We achieve the application level transparency by leveraging the dynamic linker feature of GCC. Concretely, GCC provides an option named "PRE_LOAD" which allows users to selectively override functions in shared libraries. The new memory allocation function *expand_malloc* needs to be compiled first into a shared library ended with .so and then the applications will be executed by dynamically linking to this shared memory. After that, all the *malloc()* in the application will be automatically replaced by *expand_malloc()*.

The implementation of *expand_malloc()* needs to determine when it should resort to the default *malloc()* and when it should allocate memory from the shared memory pool. This is critical since if the current physical memory in the VM is not enough, the shared memory allocation should be triggered on demand. Concretely, when a *expand_malloc()*

has been called by the application, it first checks whether the usage of the VM swap area has been increased. If not, it implies that there is still enough memory in the VM, thus *expand_malloc()* will invoke the original *malloc()* from glibc to allocate memory from the guest OS in a traditional way. Otherwise, an increasing utilization of swap partition is observed, which indicates that the VM may be short of memory. If we keep allocating memory via the default *malloc()*, more severe VM memory swapping could happen, and the VM performance may fall through the floor (seriously degraded). Therefore, *expand_malloc()* should call the *shm_malloc()* to allocate memory from the shared memory region. As shown in the experiments section, with memExpand, the VM memory can be expanded on demand without overhead of monitoring working set, while significantly reducing the amount of memory swapping traffic.

Remarks. In the first prototype implementation, the memory region allocated by the host is shared among multiple VMs, and each VM can locate only the data in its own shared memory chunks through metadata such as *list_descriptor* and *chunk_descriptor*. A malicious attacker may eavesdrop or manipulate the data of other VMs. One approach to mitigate this risk is to use *VM grouping*. By allowing users to put VMs that have mutual trust into a trust group, MemLego can allocate for each VM trust group a different shared memory region from the host machine. Therefore, VMs in one group cannot access the shared memory pools used by VMs in other groups. Another complimentary mechanism is to audit the access to the shared memory chunks to constrain a VM to access only its own *list_descriptor* and *chunk_descriptor* metadata through metadata encryption. This eliminates risks of any targeted attack due to eavesdroping or malicious manipulation.

5.3.4 Memory Swapping Optimization

We have shown in Figure 24 that VM memory swap happens even when there is enough free memory at the host and when a workload starts swapping, performance degrades drastically. One approach to significantly improve performance in the presence of memory swapping is to provide a shared memory based swap facility, which reads and writes swap pages to the shared memory swap partition in the host memory. This motivates us to develop *MemSwap* as an integral part of the MemLego for swap optimization. The main novelty of MemSwap is two folds. First, it provides a hybrid memory swapping model, which treats a fast but small shared memory swap partition as the primary swap area whenever it is possible, and *strategically* transits to the conventional disk-based VM swapping on demand. Second, it provides a fast swap-in optimization, which enables the VM to *proactively* swap in the pages from the shared memory using an efficient batch implementation.

MemSwap shares similar motivation and assumption as Frontswap [6], a Linux kernel patch, which uses the transcendent memory (tmem) as a VM swap space. Both are motivated by the common goal of improving swap performance using idle memory. Both require the presence of sufficient idle memory to be effective. In Frontswap, a hypercall has to be invoked for each swapped out page, and the swap-in operations depend on the page faults, which are costly. Given that Frontswap is currently working only on Xen. To compare with *MemSwap*,we implement Frontswap on KVM and report our performance comparison result in Section 5.4.

Hybrid swap-out. MemSwap employs a hybrid VM swap model by using both shared memory and the disk, to handle the situation when the size of shared memory is not large enough to hold all the swapped pages from a VM.

When the shared memory pool for swap is reaching a pre-defined capacity, MemSwap will trigger the hybrid swap process. We propose a least recent pages to disk approach, which puts older pages to the disk swap area when the shared memory swap partition is full, enabling the most recent VM swapped out pages to be kept in the shared memory. One way to organize the shared memory swap partition is to use a ring buffer. We maintain a *shm_start* pointer and a *shm_end* pointer pointing to the swapped pages with the smallest and largest offset in the buffer respectively. If a page fault comes with an offset between these two pointers, all the accesses can be served from the shared memory. Otherwise, the conventional disk based swap path will be invoked. In this design, a separate working thread is standing by and ready to be triggered to flush the pages from the shared memory to the disk swap area. In order to parallel the disk I/O and page swap operations, the working thread starts flushing the page when the shared memory is partially full, say $m\%$ full.

Proactive swap-in. Traditional OS provides two mechanisms to swap in pages from the disk swap area: *page faults* and *swapoff*. It is known that relying on the page faults to swap in the pages from the swap disk is expensive, and sometimes results in unacceptable delay in application performance recovery. Therefore, *MemLego* implements the proactive swap-in by extending the *swapoff* mechanism. Concretely, *swapoff* is an OS syscall that enables to swap-in pages from the disk swap area in a batched manner. However, directly applying *swapoff* will not provide the speed-up required for fast application performance. Since in order to swap in a page X, which has a corresponding entry PTE (X_pte) in the page global directory (PGD), the OS has to start from the PGD, and traverse through all the PUD, PMD, and compare with all the PTEs until the X_pte is found.

Our idea for an efficient implementation of *proactive swap-in* mechanism is to maintain some meta data in the swap area, which can assist *MemSwap* to quickly locate the corresponding PTEs in the page table. Concretely, when a page is swapped out, MemSwap will keep the address of the corresponding PTE as the metadata in front of the swapped out page in the shared memory. For each page of 4KB, its meta data only takes up 4 bytes. The cost of keeping this metadata in the swap area is around only 1/1000 of the total size of the swapped out pages. For those shared pages which have multiple PTEs, we allocate a specific area in the shared memory as *PTE store*. In this case, the first byte of the meta data specifies the number of PTEs related to this page, while the lasts three bytes is an index pointing the first related PTE in the *PTE store*. When a page is swapped in, *MemSwap* is able to quickly locate the PTE(s) that needs to be updated by referring to this metadata without the need to scan the page table. Thus, the time spent on accessing the PTE of a page to be swapped in from the shared memory is only one time memory access, and it will not increase as the size of the system wide page table grows.

5.3.5 Inter VM Communication Optimization

MemPipe is a shared memory based inter-VM communication optimization implemented on top of *ShmManager* in MemLego. If two VMs on the same hosts (co-located VMs) want to communicate with each other via network, MemPipe will be invoked and creates a piece of shared memory by using the interfaces provided by ShmManager. Network packets among co-located VMs will be transferred through the shared memory. There are several advantages of utilizing shared memory via MemPipe over the traditional network communication: (1) The communication path is shorter by using the shared memory. Since packets will be intercepted inside the VM and redirected to the shared memory, the network stack in both VMs can be skipped. (2) Since the shared memory establishes a channel for inter-VM communication, the hypervisor does not need to be involved to help transfer the packets, thus the context switch between the VMs and the hypervisor can be avoided. (3) The network packets do not have to be copied between the VMs and the host.

There are two design choices for implementing co-located VM detection mechanism: *centralized* and *decentralized*. The centralized method periodically collects the status from VMs co-located on the same host and thus introduces delayed detection and updates. Alternatively, the decentralized mechanism is event-driven. When a VM is deployed or migrates in or out of a host machine, the VM notifies the co-located VMs and the co-location information is updated synchronously upon the occurrence of the corresponding events. The decentralized mechanism does not require the involvement of the host and provides more fresh and consistent VM co-location information. We implement the decentralized mechanism in MemPipe.

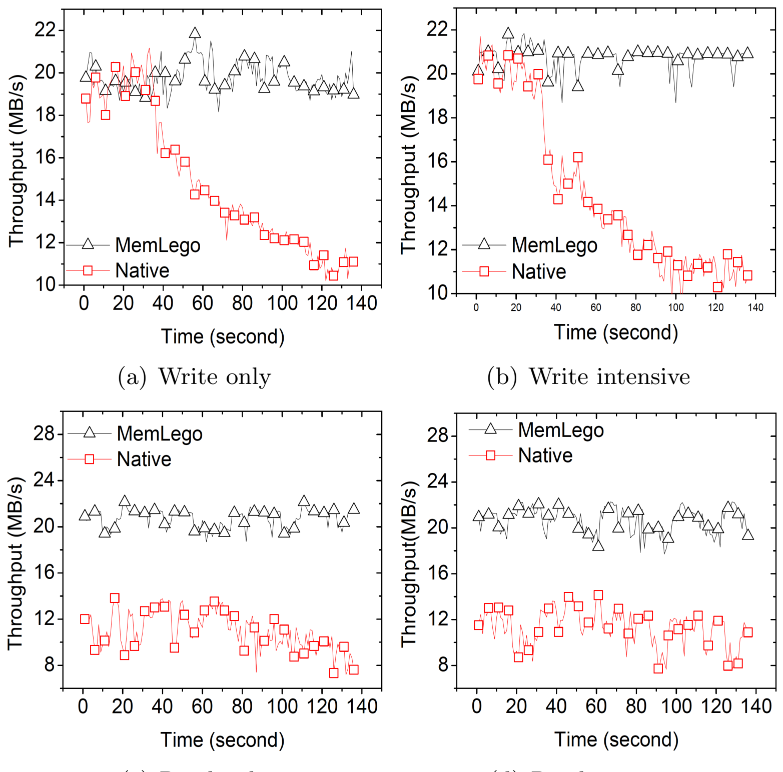
MemPipe also introduces techniques, such as *socket buffer redirection* and *anticipated time window* (ATW) to further improve the performance of inter-VM communication. The former allows the sender VM's packets to be directly copied from the user space to the shared memory, skipping the VM kernel buffer. The latter optimizes the per packet notification by a ATW based notification grouping with bounded delay by tuning time window and batch size, which can effectively reduce the number of notifications between sender VM and receiver VM, and significantly cut down the amount of software interrupts to be handled in both sending and receiving VMs.

5.4 Evaluation

The experiments are conducted on an Intel Xeon based server provisioned from a SoftLayer cloud [24] with two 6-core Intel Xeon-Westmere X5675 processors, 24GB DDR3 physical memory available for the guest VMs, 1.5 TB SCSI hard disk, and 1Gbit Ethernet interface. The host machine runs Ubuntu 14.04 with kernel version 4.1.0, and uses KVM 1.2.0 with QEMU 2.0.0 as the virtualization platform. The guest VMs also run Ubuntu 14.04 with kernel version 4.1.0. 4 VMs are simultaneously running on the host. We measure the performance of MemLego by using the following applications and benchmarks: (a) Redis[22], an open source, in-memory key-value store, which supports a wide spectrum of data structures such as strings, hashes, lists, bitmaps, and is often used as database, cache, and message broker; (b) MemCached [12], a distributed hash table based in-memory key-value store for small chunks of arbitrary data (strings, objects), such as database calls, API calls, or page rendering; (c) MapReduce [61]; (d) Netperf [14], a benchmark that can be used to measure the performance of many different types of networking. It provides tests for both unidirectional throughput, and end-to-end latency; (e) OSU MPI benchmarks [17]; and (f) SCP and Wget, Linux commands to transfer files between two machines through the network.

5.4.1 MemLego with MemExpand

We first measure the effectiveness of MemLego core with only MemExpand turned on but MemSwap and MemPipe disabled. We show how MemLego can reduce and eliminate the performance degradation of applications running on the VMs under memory pressure (as those shown in Figure 1). We use Redis [22] and Memcached [12] as example applications. We use YCSB [53] to generate the client workloads for Redis, and Memtier [13] to create the client workloads for Memcached since YCSB does not support Memcached. In this set of experiments, 4 VMs are running on the host machine, and each is initialized with 4GB memory and 4GB disk swap area. VM1 is working as the Redis server or Memcached server for Redis or Memcached measurements respectively, and the other 3 VMs are idle initially. We measure the performance of the server VM with four types of workloads: write only, write intensive, read only, and read intensive. The first two workloads inject 6GB data



(a) Write only (b) Write intensive

(c) Read only (d) Read intensive

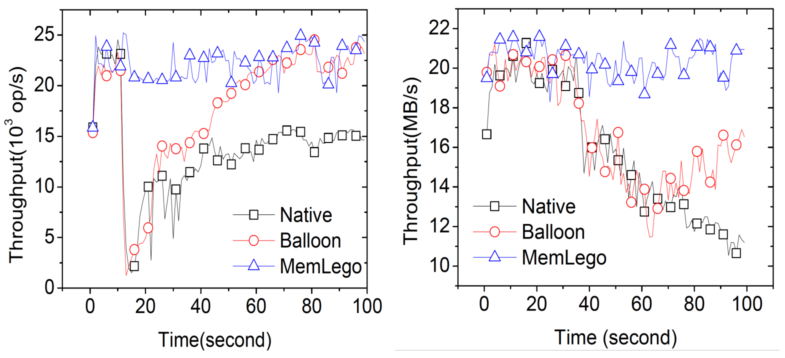
Figure 41: Effectiveness of MemLego on Memcached

into the server VM, while the latter two workloads uniformly read the injected data. The read-write ratio is 100:1 in the read intensive workloads and ,1:100 in the write intensive workloads.

Figure 41 shows the effectiveness of MemLego for the 4 typical types of workloads on Memcached. We make two observations. First, for write only and write intensive workloads, MemLego and native system start with similar performance at the beginning, because VM1 has sufficient memory and every write operation corresponds to a single memory access. However, around the 33th second, the performance of the native case starts to drop drastically. This is because there is insufficient memory in VM1 to continue its write operations, which leads to increased memory swapping traffics. Many write operations need disk-IO due to the need for making room for the new data to be written to Memcached and consequently the increased swap-out/swap-in paging to the disk swap partition. In comparison, MemLego handles such surge of memory pressure calmly and smoothly. The performance of both write only and write intensive workload is not affected much by the memory pressure experienced in VM1 and the addition of more memory from the shared memory pool to VM1 by MemExpand. For the write only workload, its throughput stays the same, around 20MB/sec.

Second, MemLego shows the improved performance of the read only and read intensive workloads over the native case as well. For the read intensive workload, MemLego improves its throughput by 75%, from around 12MB/sec to 21MB/sec. In the native case without MemLego, the 6GB data bulk loaded to Memcached server (VM1) with only 4GB DRAM. Thus, about 2GB data is put into the 4GB swap disk mounted to the VM1 in native case while in MemLego, this additional 2GB data is put into the shared memory swap partition instead. Thus, for read workloads with uniformly reading over the 6GB data, some read requests may have to fetch data from the disk through swap-in paging, which incurs much higher overhead due to the slow disk IOs involved. This results in that the performance of the read operations in the native case is consistently worse than that of MemLego. In addition, we show again for the read only and read intensive workloads that there is no visible performance overhead for the server VM to read data from the shared memory swap area. Similar results are observed from performing the same set of experiments by running Redis server on VM1. Due to space constraint, we omit them here.

Next, we compare the performance of MemLego to that of the native system with the balloon driver enabled using both Redis and Memcached. As suggested by [120], the balloon driver is triggered every 30 seconds to balance the memory from high pressure VMs to the low pressure VMs. Figure 42 shows similar trends for both Redis and Memcached workloads: The Native case performs the worst due to high page swapping traffic under memory pressure. Balloon driver enabled native case improves the performance as soon as ballooning is triggered (see the middle curve), with the throughput of Redis starting to recover after 30^{th} seconds and the throughput of Memcached starting to recover around 60th second. In comparison, MemLego is able to keep the performance of both Redis and Memcached around the peak performance stably, even when VM1 is under memory pressure and receiving additional memory from the shared memory pool by MemExpand.



(a) Redis (b) Memcached

Figure 42: Effectiveness of MemLego v.s. Balloon driver

5.4.2 MemLego with MemSwap

We measure MemLego performance with MemSwap enabled from three perspectives: (1) We explore how MemLego can be affected by the settings of guest OS parameters; (2) We measure the performance of MemLego with MemSwap by running Redis workloads; and (3) We compare the performance of MemLego with MemSwap with native system powered by Frontswap. To better understand the unique contribution of MemSwap to the VM performance improvements, all the experiments are performed by using MemLego with MemSwap enabled and MemExpand and MemPipe disabled.

We evaluate the effectiveness of MemLego with MemSwap by comparing its performance on Redis workloads with the native system. In this set of experiments, 4 VMs are running on the host, and each VM is initialized with 4GB memory and 4GB disk swap area. A Redis server is running on VM1, which is pre-loaded with 5GB data in memory, while the other 3 VMs are idle. Different workloads generated by YCSB [53] are executed by a remote client on VM1. We let the client run for about 10 second, and then start another memory intensive applications using *memAlloc* on the same VM1 where the Redis server is running, which allocates 7GB memory and causes the total memory demand of VM1 to increase to 12GB. For the case of the native system with Balloon enabled, the balloon driver will move 2GB memory from each of the remaining 3 idle VMs to VM1, while in the MemLego case, the balloon driver will also move a total of 6GB from the other 3 VMs with 5GB to VM1 and 1GB to the shared memory swap area used by VM1 in the host. Also for the 5GB

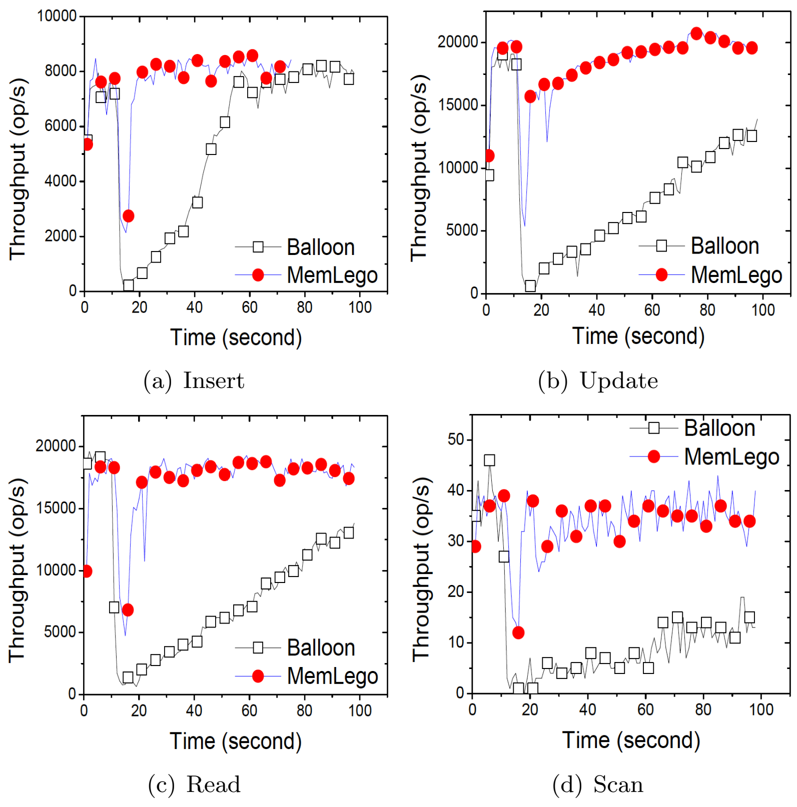


Figure 43: Throughput of Redis server measured by YCSB workloads

to VM1, 2GB memory from VM2 and 2GB from VM3, and 1GB from VM4. We measure the throughput as well as the latency of the Redis server, and compare the Redis server performance for the two systems: (a) MemLego with MemSwap enabled but MemExpand disabled and (b) native system with Balloon driver enabled. The workloads used in this set of experiments include *Insert, Read, Update, and Scan*, and are generated by YCSB with a uniform request distribution.

Figure 43 displays the throughput results. First, the Redis throughput drops significantly at around the 10th second for both systems, no matter which workload is running. But the performance drop is smaller and the performance recovery is faster for MemLego. This is due to the face that there is insufficient free memory in VM1 when the second memory intensive application starts. However, for native case, VM1 cannot get free memory from the balloon driver immediately, which causes memory swaping in VM1, and increased swapping leads to serious degradation of the Redis performance. In comparison, for MemLego case, the performance drops sharply at the 10th second but less serious than the native system. This is because MemExpand is disabled and MemSwap has only 1 GB in the shared memory swap area. As soon as MemLego gets sufficient memory from other three VMs, the Redis performance recovers immediately for all 4 types of workloads. In contrast, for the native system with Balloon case, the performance of all workloads are slowly recovering due to the slow page-fault based swap-in and high cost of disk IOs to access the disk swap partition. Both page fault and disk I/O are very expensive operations. For example, consider the *Read* workload in Figure 43(c), its throughput drops from 17813 OP/sec to 7049 OP/sec at the 10th second, but it only takes about 5 seconds for the throughput to be fully recovered in MemLego, whereas it took more than 80 seconds for the Native system with Balloon case to recover to the previous peak throughput performance.

5.4.3 MemLego with Different Configs

In this section we evaluate MemLego with three settings: MemExpand enabled, MemExpand and MemSwap both enabled, and MemLego with MemExpand, MemSwap and MemPipe all enabled. We measure how much performance improvement that each of the three components can contribute to the overall effectiveness of MemLego. Two VMs are running on the same host. Each VM is initialized with 4GB memory and 4GB disk swap area. Also 3GB host memory area is reserved and shared between the VMs, which is equally divided into three regions (1GB for MemExpand, 1GB for MemSwap, 1GB for MemPipe). One of the VM is running as a Memcached server, while the other VM is running as a client. The write only and write intensive workload will inject 6GB data to the Memcached server, while the read only and read intensive will read uniformly over these data.

Figure 44 shows that for all 4 workloads, their performance is gradually increasing as we add one more component into the MemLego system. For the write only workload, the Native case has the worst performance. The reason is that the Memcached server (VM1) has 4GB memory in total, with 6GB data being loaded. Thus 2GB data needs to be swapped out, which seriously degrades the performance of Memcached server. Compared to the *MemLego* case with only MemExpand enabled, it can improve the average throughput of the write only workload from 8MB/s to 14MB/s. Turning on *MemSwap* further improve the performance by 50%, since by adding *MemSwap*, another 1GB of the shared memory can be used by the server VM as its swap area. Finally, by integrating *MemPipe*, the throughput of the write intensive workload can be improved from 21MB/s to 27MB/s, since *MemPipe* accelerate the data transfer between the client VM and the server VM co-located on the same host.

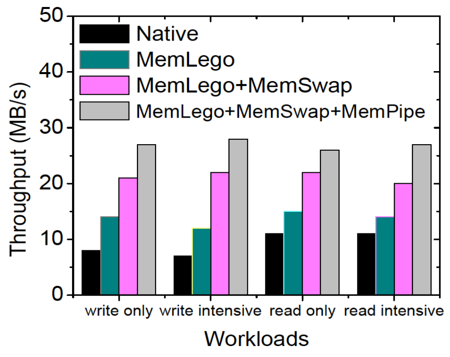


Figure 44: Performance of Memcached under different MemLego configurations

5.5 Conclusions

We have presented the design and implementation of MemLego, a shared memory based memory optimization framework for managing and improving memory efficiency in virtualized environment. MemLego makes three original contributions: It offers on-demand VM memory allocation and deallocation in the presence of changing workloads. It relieves VM execution performance from drastic degradation due to memory swapping. It provides shared memory pipes for high-performance communication between co-resident VMs.

Chapter VI

STACKVAULT: PREVENTING SENSITIVE STACK DATA LEAKAGE FROM UNTRUSTED THIRD PARTY FUNCTIONS

Recently, data exfiltration attacks via RAM scrapping have led to huge data breaches such as that of Target and Neiman Marcus (70M records). Such attacks exploit the fact that sensitive data, often unencrypted, are stored in process memory. This chapter presents *StackVault*, a kernel backed system level facility to prevent sensitive data on the stack from being accessed in an unauthorized manner by an untrusted third party function. *StackVault* developed a novel and unforgeable function identity to prevent an untrusted function to steal data from a protected stack. *StackVault* has a three-phase framework in order to secure sensitive data on the stack: (1) capturing application-specific sensitive functions and untrusted functions through easy-to-use configurations, (2) transparent placement of stack protection operations through system-supplied secure procedures to protect, restore, and clear the stack, and (3) automated enforcement of stack protection through spatial and temporal access monitoring and control over both sensitive stack data and untrusted functions. We have evaluated *StackVault* using a number of popular real world applications. Our results show that *StackVault* is effective and efficient, while incurring negligible performance overhead for applications in most scenarios.

6.1 Introduction

Using the third-party libraries is a common approach to facilitate the software development. However, third-party libraries are also becoming one of the most *insecure* components of an application [28]. For example, the Heartbleed [33] is a known security bug in the OpenSSL cryptograph library. The compromised library can read more data than it should be allowed. According to [3], the usernames and passwords of many US bank customers have been leaked due to the third-party scripts integrated in the login pages. Recently, a GUN C library has been identified as vulnerable [35], showing that even some of popular and widely used third party library can be untrusted. Therefore, preventing sensitive data leakage in the presence of such untrusted APIs is becoming an important security challenge.

One of the vulnerabilities that an untrusted third party API can cause is the data leakage attack to the stack frame, which is an essential component of a function during the runtime. It contains local variables, function arguments, return addresses and so forth. By design, each function has its own stack, the range of which is specified by the CPU registers (e.g., RSP and RBP on the x86_64 platform). A well-behaved function is not expected to access the data on the stack of another function, except when chasing a pointer. However, all functions in a single program are running in the same address space, thus stack layout is quite predictable and easy to guess. Furthermore, currently there is no mechanism to restrict one function from accessing another function's stack if they are in the same process. Thus, illegal stack access can result in sensitive data leakage.

State of the art research efforts. Existing research efforts on protecting stack data leakage have centered on two most well-known vulnerabilities: the uninitialized read problem and the stack overflow attack. Examples of the uninitialized read attack have been reported by numerous companies, such as Microsoft [32], and open source software, such as Samba [34]. The most recent advance on this problem is represented by SafeInit [98]. It promotes a compiler based optimization to address this problem by explicitly initializing each variable right after its allocation. Such vulnerability has also been identified by [93] in Linux kernel. Other mechanisms such as ASLR [116] and StackArmor [49] rely on randomization to protect the data on the stack in a probabilistic manner. StackGuard [49] addressed the stack overflow problem by inserting a canary word next to a return address such that if the canary word is damaged, stack corruption can be detected. StackShield [26] copies valid return addresses to a safe memory location that cannot have overflow, and check them before the function returns. Some efforts propose to slightly modify CPU hardware to protect against stack smashing. For instance, StackGhost [67] uses SPARC CPU hardware to get OS in the loop to armor the stack. This line of efforts has the advantage that there is no need to recompile code but it suffers limited adoption since it needs specific hardware. Orthogonal to the efforts on stack data leakage, recent research on trusted computing platform puts forward a commodity solution, represented by SGX [54], which uses enclaves to protect selected code and data from being leaked. But it does not consider the case in which the code inside the enclaves is untrusted or compromised. Little efforts have been made on protecting the sensitive data on the stack from untrusted functions, such as third party APIs.

Problem statement and solution. In this chapter, we argue that in addition to the attacks considered above, an attacker may exploit other channels to hack the stack data. For example, a function has access to the stack of the whole process it belongs to, and thus it can look beyond its own stack frames. Sensitive data can be leaked when an untrusted function is invoked while stack frames of the functions holding sensitive data are already present in the process stack. Specifically, since RSP and RBP are two general registers on the x86_64 platform that can be modified from user level, an attacker can simply obtain the stack boundaries from the RSP and RBP registers and scan the stack memory accordingly. In this case, the attacker can readily bypass the existing state of art approaches to gain access to the sensitive data on the stack, in addition to reading from the uninitialized variables and compromising return addresses to launch the stack overflow attack. Programmers can suffer from such sensitive stack data leakage attacks when they are using untrusted functions in third-party libraries. Since these libraries are not fully developed by the users, and are under the risk of being compromised. When a user invokes a compromised library call within a function or after a function, which contains sensitive data on its stack, the sensitive data can be disclosed.

In order to solve this problem, we introduce *StackVault*-a kernel-backed system-level facility to eliminate sensitive stack data leakage. This work is motivated by real applications scenarios that software development team at many companies face when writing large-scale applications with sensitive data [20]. Examples of such data include protected health information(PHI) and financial records. Concretely, *StackVault* enforces three types of stack protection operations to protect the sensitive stack data by preventing an untrusted function from illegally accessing the stack of another function in the same process. Through placement and enforcement of such operations, *StackVault* moves the sensitive stack data into an OS kernel buffer prior to the execution of an untrusted function, so that there is no way for such data to be touched by any untrusted function. Such protection also ensures that all data required for execution of the untrusted function is kept on the stack. The stack data is restored immediately after the untrusted function returns, and the stack is cleared for every sensitive function upon its return, in order to eliminate any leakage of stack data after its completion. We assume that the OS kernel is the trusted computing base, and any buffer in the kernel cannot be accessed by the attacker in the user level.

StackVault prevent stack data leakage through a three-phase stack protection framework: configuration phase, compilation phase and runtime phase. This design is highly transparent and does not require any source code modification. In the configuration phase, users only need to specify the names of the functions that contain sensitive data on their stack, and the names of the functions that are considered untrusted. In the compilation phase, *StackVault* injects protection operations into the object code to prevent data leakage attacks. In the runtime phase, a kernel level module is employed to provide a trusted data protection area (kernel buffer) with safe access enforcement. We have introduced a novel and efficient notion of "identity of a function". Such identity is unforgeable and a malicious function cannot be disguised to illegally access the sensitive stack data under the protection of *StackVault*.

The design of *StackVault* addresses three challenges:

- **Correctness.** In order to maintain the correctness of the program while protecting the function stack simultaneously, extra attentions need to be paid to the function return values and the parameters residing on the stack. Put differently, while preventing illegal accesses to the function stack, the return value should not be affected and the parameters should still be accessible.
- **Unforgeable function identity.** Stack protection APIs provided by *StackVault* are to be invoked to protect specific stacks against sensitive data leakage. It is extremely important to detect and prevent any illegal invocation of such APIs by the attackers to steal data from a protected stack area. Therefore, *StackVault* needs to keep track of the functions that have invoked the stack protection API, and only accepts the API call if it is legitimate. From inside a system call, it is challenging to securely and accurately detect which user

level function invokes it, especially when there are multiple user level functions in the same process.

- **Usability.** *StackVault* should be highly transparent to the existing applications, so that an application can benefit from the *StackVault* facility without modification of its source codes. This is critical to reduce the amount of work that developers need to carry out in order to adopt our tool to their products.

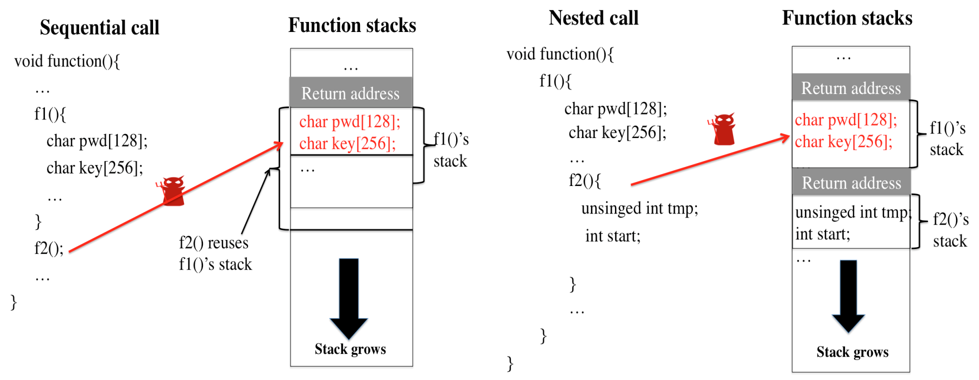
This chapter makes three original contributions. First, we identify and analyze the sensitive data leakage attacks in the presence of untrusted functions, such as third party libraries. Second, we design and implement *StackVault*, an OS kernel backed system-level facility, which is light weight and employs three phase secure architecture to prevent untrusted functions from illegally accessing the sensitive data on the stack of a sensitive function. *StackVault* is by design easy to use, and does not bring any new venerability to the system it protects against data leakage. Third but not the least, we develop an effective mechanism to securely and accurately identify, from inside a system call, which user level function invokes this system call. This technique prevents a system call from being abused by a malicious user level function during the runtime. We evaluate the correctness and effectiveness of *StackVault* using real world applications, such as *Minizip*, *Curl*, *OpenSSH*, and *Netperf*.

The rest of the chapter is organized as follows: Section 6.2 describes the threat model and motivating examples. Section 6.3 presents the design and implementation of *StackVault*. We experimentally evaluate *StackVault* in Section 6.4, discuss the related work in Section 6.5, and concludes the chapter in Section 6.6.

6.2 Threat Model and Motivating Example

Stack is a virtually continuous piece of memory that is allocated for each function to store its local variables, return address, as well as the values of registers that need to be temporarily saved. Sensitive data on the stack can be leaked in the presence of an untrusted function because an untrusted function can acceess sensitive information from the stack of other functions in the same process. Although an untrusted function can do others tricks, such as dividing zero to crash the system, in this chapter, we focus on developing a system-level facility to protect sensitive stack data from being illegally accessed by untrusted functions.

We consider the following threat model. Conceptually, each function has its own isolated stack memory. A well-behaved function is not supposed to read or write the stack memory of another function unless it needs to access some specific parameters. Given that stack memory of different functions within a program are allocated in the same address space, and there is no mechanism to prevent one function from accessing the stack of another function, an untrusted function can easily access the data on the stack of the other functions. If sensitive data, such as username and password, is allocated on the stack, such sensitive data can be leaked to untrusted functions. Third party APIs is one category of such untrusted functions, since they are not fully written in house and can be compromised by being downloaded from a fake website. In addition to third party APIs, an untrusted function can be any function whose behavior is not fully controlled by a user. For example, when multiple companies are contributing code to the same project, a function that developed by one company can be an untrusted function for another company who is using the function. We below use an example to illustrate the existence of such security threat and how it can result in sensitive data leakage.



```
void function(){
    …
    f1(){
        char pwd[128];
        char key[256];
        …
    }
    f2();
    …
}
```

```
void function(){
    f1(){
        char pwd[128];
        char key[256];
        …
        f2(){
            unsinged int tmp;
            int start;

        }
        …
    }
}
```

(a) Sequential: f2() is invoked after f1() (b) Nested: f2() is invoked within f1()

Figure 45: Stack layout of two functions when they are invoked in a sequential or a nested manner.

Figure 45 displays the stack layout of two functions *f1()* and *f2()* when they are invoked in either a sequential or a nested manner. We assume that *f1()* is a function that has sensitive data on its stack, while *f2()* is an untrusted function. In the nested case, the untrusted function *f2()* is invoked within *f1()*. Since the sensitive data of *f1()* still resides on its stack during the execution of *f2()*, *f2()* can easily get these data if it is compromised. In the sequential case, *f1()* allocates two pieces of sensitive data – password and key – on its stack, but does not clear them before it returns. Thus, when the untrusted function *f2()* starts to run, it will be able to access these sensitive data on the stack of *f1()*.

Listing 6.1 illustrates a piece of code which uses libcurl APIs to upload a file to a specific URL. After uploading the file, the *do_others()* function will be invoked in *main()*. The local variable *struct stat file_info* is allocated on the stack of the *uploadfile()* function. It is a system defined data structure and its definition is given in in Listing 6.2. Clearly, *struct stat file_info* carries critical information of a file, such as the user and group ID of the file, the file serial number, and the time of last access of this file. Such information is often considered as sensitive data if the file is a private or sensitive file.

```
int uploadfile(FILE *fd, char *URL)
{
  CURL *curl;
  CURLcode res;
  struct stat file_info;
  double speed_upload, total_time;

  fd = fopen("debugit", "rb");
  if(!fd) {
    return 1; /* can't continue */
  }
  /* Initialize file_info */
  if(fstat(fileno(fd), &file_info) != 0) {
    return 1; /* can't continue */
  }
  curl = curl_easy_init();
  if(curl) {
     /* upload to this place */
    curl_easy_setopt(curl, CURLOPT_URL,
    URL);
    ...
    curl_easy_setopt(curl,
    CURLOPT_INFILESIZE_LARGE,
24        (curl_off_t)file_info.st_size);
25        ...
26        }
27        else {
28          ...
29        }
30      }
31      return 0;
32    }
33
34    void main(void){
35        FILE *fd;
36        char *URL;
37        ...
38        uploadfile(fd, URL);
39        do_others();
40    }
```

Listing 6.1: Using libcurl to upload a file to a specific URL

As shown in Listing 6.1, the variable *struct stat file_info* is initialized at line 13, and accessed at line 22. The libcurl APIs are invoked at line 13, line 16, line 19, and line 22. The invocation at line 16 and line 19 are not supposed to access the *file_info* structure. However, as indicated in Figure 45(b), since the data on the stack of the *uploadfile()* function is still accessible during the execution of all these libcurl APIs, if either *curl_easy_init()* or *curl_easy_setopt()* has been compromised, they will be able to access the *file_info* structure without any difficulty. Also, in the main program,the *do_others()* function is executed immediately after the *uploadfile()* function and thus can illegally access the *file_info* structure if being compromised. Both cases result in the leakage of sensitive data on the stack of the *uploadfile()* function.

At the same time, even though the *curl_easy_setopt()* is allowed to access the *st_size* field of the *file_info* structure, it is not safe to allow the other parts of the *file_info* to be accessed by *curl_easy_setopt()*.

Besides libcurl, there are many other open source libraries such as libssh, libpcap, openssl, which are popularly used in software development. Users of such libraries are under the risk of disclosing sensitive data, due to either the lack of test of these libraries or downloading them from a compromised website.

```c
struct stat {
    dev_t     st_dev;/* ID of device
                containing file */
    ino_t     st_ino;
    mode_t    st_mode;
    nlink_t   st_nlink;
    uid_t     st_uid;/* user ID of owner */
    gid_t     st_gid;/* group ID of owner */
    dev_t     st_rdev;
    off_t     st_size;
    blksize_t st_blksize;
    blkcnt_t  st_blocks;
    struct timespec st_atim; /* time of
                        last access */
    struct timespec st_mtim;
    struct timespec st_ctim;
    #define st_atime st_atim.tv_sec
    #define st_mtime st_mtim.tv_sec
    #define st_ctime st_ctim.tv_sec
};
```

Listing 6.2: Definition of struct stat

6.3 System Design

We make the following efforts to ensure the correctness and security of *StackVault*. First, *StackVault* introduces three procedures to protect the sensitive data on the stack of a function, so that it cannot be accessed by am untrusted function. Second, *StackVault* develops a novel notion of unforgeable "function identity" so that an untrusted function cannot masquerade as a normal function to invoke the *StackVault* procedures. Third, *StackVault* guarantees that an untrusted function can still access its parameters on the protected stack frames.

6.3.1 System Overview

We design and implement *StackVault* as a system-level facility for protecting sensitive stack data from untrusted functions. We refer to those functions that have sensitive data in their stack as sensitive functions, e.g., *f1()* in Figure 45. Our attack model has identified two types of relationships between a sensitive function (e.g., *f1()*) and an untrusted function (e.g., *f2()*), which can cause sensitive stack data leakage.

The sequential case is the simple one where the untrusted function is sequentially following the sensitive function. To prevent the sensitive stack data leakage in this case, we introduce a function named *clear_stack()*, which clears the stack of a sensitive function such as *f1()* after its execution. This *clear_stack()* operation makes sure that functions executing after *f1()* will not be able to obtain data on the stack of *f1()*.

The nested relationship is the complex case where the untrusted function is invoked within the sensitive function. Such nested relationships can be direct or indirect. For example, recall Figure 45, the sensitive function *f1()* and the untrusted function *f2()* have a direct nested relationship. In *StackVault*, we introduce two other system calls *start_protect()* and *end_protect()* to protect state data leakage in this second case. For example, by jointly using these two *StackVault* operations, we exercise the stack protection over the *f1()'s* stack before *f2()* is invoked and release the protection on the stack after *f2()* returns.

The main idea of *StackVault* is to hide the stack data of a sensitive function in a secure place, prior to the execution of an untrusted function in its nested call structure. Specifically, *StackVault* uses a kernel buffer to hide the stack memory of a sensitive function. We choose to use the kernel buffer for two reasons. First, the kernel buffer created by *StackVault* is considered not accessible by user-level functions, including the untrusted functions. Second, we also provide additional guards to prevent attempts to misuse and abuse of *start_protect()* and *end_protect()* by leveraging the three-phase stack protection framework of *StackVault*: Configuration, Compilation and Runtime Verification.

In the configuration phase of *StackVault*, users need to provide two lists: a *sensitive function list* and an *untrusted function list*. *Sensitive function list* includes names of the functions that contain sensitive data on their stack. Which data is sensitive highly depends on the users. The *untrusted function list* consists of the names of the functions whose behaviors cannot be fully trusted by the users, such as library calls or some third party APIs, which are not entirely developed by the users. *StackVault* will prevent the untrusted functions from accessing the stack memory of the sensitive functions. In the first prototype of *StackVault*, we ask the user to put this configuration file in the root directory of their source code. Also we require the users to compile their source code using the provided LLVM [86]. Note that users do not need to change their source code in order to use *StackVault*.

In the compilation phase of *StackVault*, the invocation relationship between an untrusted function and a sensitive function is discovered. Based on the type of such relationships, *StackVault* inserts the right choice of *StackVault* operators, which will enforce the execution of the adequate stack protection operations before or after an untrusted function call.

Before we engage in more detailed design of our compilation phase and our runtime verification phase, we first illustrate these two phases using an example. In this example, *pwdgenerator()* is a sensitive function and *lib_func()* is an untrusted function as shown in the pseudo code in Listing 6.3 (only the lines colored in black). Figure 46 shows the interaction between configuration phase and compilation phase. The executable files are generated by the *StackVault* annotated LLVM intermediate code as output of the compilation phase.

```
void main(){
  ...
  /*pwdgenerator is a trusted function*/
  pwdgenerator(){
      /*sensitive data on stack*/
      char passwd[256];
      char key[256];
      ...
      //INSERT start_protect("lib_func")
      lib_func(); /*lib_func() is
                  an untrusted function*/
      //INSERT stop_protect()
      ...
  }
  //INSERT clear_stack("pwdgenerator");
  ...
}
```

Listing 6.3: Code snippet annotated by *StackVault*

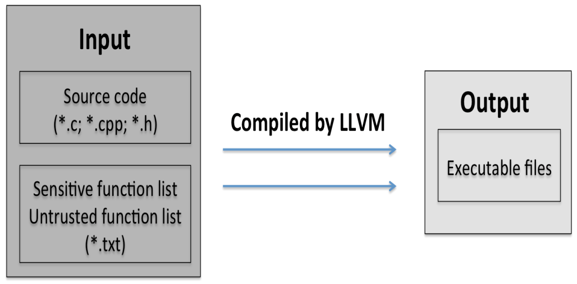
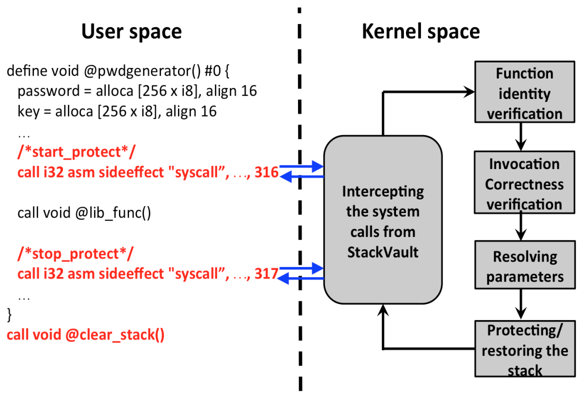


Figure 46: *StackVault* workflow

Listing 6.3 gives a pseudo code snippet showing where and what have been annotated automatically by *StackVault* after the compilation phase (see the blue-colored comment lines). In this example, the C code is used to make it easier to understand the operators inserted by *StackVault*, since the *StackVault* APIs are actually inserted into the intermediate code generated by LLVM. From Listing 6.3, we observe that to protect the stack of a sensitive function in the presence of untrusted function *lib_func()*, three *StackVault* system calls are inserted to LLVM generated code:

- **start_protect(char *func_name)**. This is a system call which protects the stack of the function that invokes it. This system call takes as the input parameter the name of an untrusted function and is placed prior to the invocation of this untrusted function. It figures out which areas of the stack should be kept accessible after protection, since they are the parameters of the untrusted function with name *func_name*.
- **stop_protect(void)**. This is a system call which restores the stack of the function that invokes this system call. It is used in pair with *start_protect(char *func_name)*.
- **clear_stack(char *func_name)**. This is a procedure that needs to be invoked after each sensitive function to clear its stack memory. The parameter refers to the name of the function that needs to clear its stack.

Figure 47 describes interaction between the user level system call and the *StackVault* kernel module. The code snippet on the left is correspondent to that in Listing 6.3, but


User space
Kernel space
Intercepting the system calls from StackVault
Function identity verification
Invocation Correctness verification
Resolving parameters
Protecting/ restoring the stack

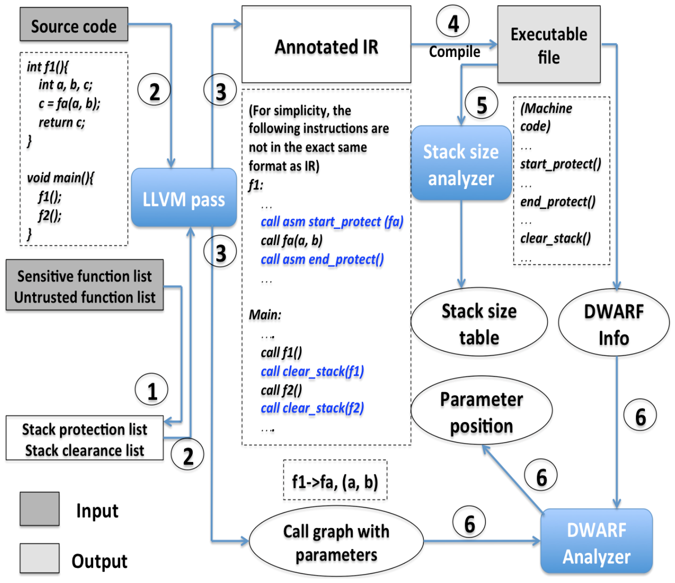
```
define void @pwdgenerator() #0 {
  password = alloca [256 x i8], align 16
  key = alloca [256 x i8], align 16
  ...
  /*start_protect*/
  call i32 asm sideeffect "syscall", ..., 316

  call void @lib_func()

  /*stop_protect*/
  call i32 asm sideeffect "syscall", ..., 317
  ...
}
call void @clear_stack()
```

Figure 47: Interaction between the system calls and the kernel module of *StackVault*

shown as the LLVM intermediate code. Two *StackVault* system calls (#316 and #317) are invoked before and after the *lib_func()*. The *StackVault* first intercepts the system call, then it checks whether the system call is invoked legitimately in terms of *StackVault* security compliance. After the compliance verification, if the system call is *start_protect()*, *StackVault* first figures out the addresses of the stack frames that need to be protected, and then hide the sensitive data on those stack frames in the kernel buffer. If the system call is *stop_protect()*, *StackVault* will locate the appropriate stack frames that are hidden in the kernel buffer and restore them back to the stack.



```
int f1(){
  int a, b, c;
  c = fa(a, b);
  return c;
}

void main(){
  f1();
  f2();
}
```

```
(For simplicity, the
following instructions are
not in the exact same
format as IR)
f1:
  ...
  call asm start_protect (fa)
  call fa(a, b)
  call asm end_protect()
  ...

Main:
  ...
  call f1()
  call clear_stack(f1)
  call f2()
  call clear_stack(f2)
  ...
```

```
(Machine
code)
...
start_protect()
...
end_protect()
...
clear_stack()
...
```

Figure 48: *StackVault* architecture

Figure 48 illustrates the internal mechanisms of *StackVault*. Based on the user-supplied lists in the configuration file and by examining the call graph of the source code generated by LLVM compilation, *StackVault* identifies the relationship between each sensitive function and each untrusted function, and derives the *stack protection list* and the *stack clearance list*. The *start_protect()* and *stop_protect()* will be inserted before and after all the functions in the former lists, while the *clear_stack()* will be inserted after all the functions in the latter list.

However, when a sensitive function has an indirect nested invocation relationship to an untrusted function, the decision on where and how to insert the pair of *start_protect()* and *stop_protect()* operators becomes more complex. We defer the detailed discussion to Section 6.3.4.

To guarantee the reliability and correctness of the programs protected by *StackVault*, two types of verifications are provided by *StackVault*. Both verification procedures will be carried out during the runtime verification phase. The first type of verification is to detect and remove illegitimate invocations of *start_protect()* and *stop_protect()*. The second type of verification is to ensure the correctness of the invocation and usage of the *StackVault* APIs. In *StackVault*, two analyzers are developed to extract necessary information from the executable file for this verification. The *stack size analyzer* is responsible for finding the stack size of each sensitive function, and the *start_protect()* and *clear_stack()* procedures will refer to this information to determine the size of the stack it needs to work on. The *DWARF analyzer* is by design to figure out the exact location of each parameter of an untrusted function, and whether these parameters are on the stack of the sensitive function. The goal of *DWARF analyzer* is to keep those parameters accessible after the stack is being protected. Section 6.3.3 will describe them in detail.

6.3.2 API Design

Figure 45(a) shows how the stack memory of *f1()* and *f2()* is layed out when *f2()* is called after *f1()*. The RBP and RSP are the registers pointing to the bottom and the top of *f2()*'s stack respectively. If the sensitive data on the stack of *f1()* has not been cleared after *f1()* returns, then the untrusted function *f2()* can illegally access the sensitive data while it is running.

Since *start_protect()* and *stop_protect()* rely on the values of two CPU registers − RSP and RBP − to protect the stack, and the values of both RSP and RPB are maintaining the bottom and top of the stack while a function is running and they can be easily changed from the user level. The key challenge is how to prevent an untrusted function from intentionally modifying the values of RSP and RBP, and then illegally obtaining the data on a protected stack using *stop_protect()*.

A straightforward design of *start_protect()* and *stop_protect()*is to take RBP and RSP as the parameters of these system calls, and to protect the data between these two addresses. However, this intuitive design is neither practical nor effective. First, not every programmer is familiar with the stack layout and it is not reasonable to require programmers to know which part of the stack they are going to protect. Second, the return address of the callee function is on the stack and should not be affected, otherwise the program cannot continue running after the callee returns. Finally, using RSP and RBP as parameters will open doors to anyone to inappropriately manipulate any part of the stack memory directly by changing the values of RSP and RBP, which could further threat the execution of the applications. Therefore, both system calls need to figure out the boundaries of the stack that they are going to protect by themselves. In *StackVault*, this is achieved by referring to the kernel stack, where the values of the user level registers are pushed to before entering a system call.

6.3.3 Verification

In *StackVault*, the system calls *start_protect()* and *stop_protect()* should be used in pair only by the legitimate caller functions. Thus we need mechanisms to prevent illegal system calls and also to ensure correct execution of system calls.

Function identity verification. Without a proper mechanism, a malicious attacker can intentionally invoke the *stop_protect()* to restore the stack data that are previously protected by *start_protection()*, which can result in stack data leakage. This is because *stop_protect()* relies on the values of the RBP and RSP registers to decide the boundaries of the stack to be protected or restored, and these RBP and RSP values unfortunately can be modified by any user level program.

Although there are mechanisms introduced by the OS to protect memory, for example, *int mprotect(void *addr, size_t len, int prot)* is a function that can set the protection level of a memory address space from the calling process to be not accessible, this approach regrettably does not solve the problem. Since the untrusted function and the sensitive function are in the same process, if the untrusted function wants to illegally read the memory data that is previously protected, it can simply invoke *mprotect()* again to set the memory as readable.

To solve this problem, our solution is to let a system call identify the address of the user level instruction that invokes it, and use this address to tell which user level function has invoked this system call. We introduce two data structures for this purpose: (1) the *protection table* and (2) the *symbol table*. Both of them are maintained by *StackVault*. The *symbol table* is created before the program starts execution and then stays constant, while the *protection table* is dynamically updated based on each invocation of the stack protection system calls. This solution can prevent an untrusted function from illegally access the stack memory of a sensitive function, even if they are in the same process.

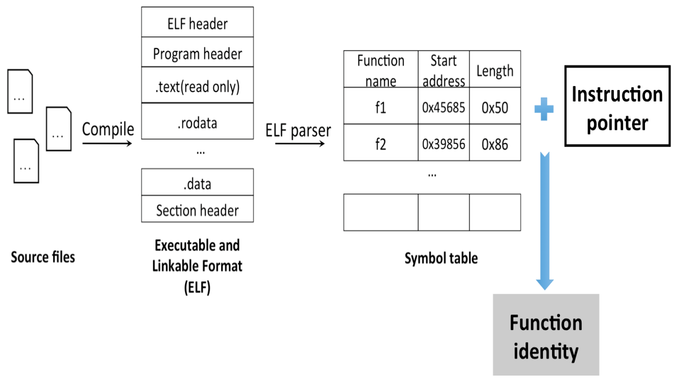


Figure 49: Details of the symbol table

Figure 49 displays the details of the *symbol table*. This table is created by parsing the .text section of the executable file of the program, which is represented in the Executable and Linkable Format (ELF). It contains three columns: function name, start address, and length. It lists the range of the instructions for each function in the program. At the same time, a system call can get the user level instruction pointer from the kernel stack. This pointer points to the address of the user level instruction right before the invocation of the system call. Therefore, given an instruction address, *StackVault* can tell which user level function is currently running by referring to the *symbol table*. This is how *StackVault* automatically identifies the function invoking the system call. The values in the *symbol table* and the instruction address are guaranteed to be secure for a number of reasons. First, since the *.text* section is a read-only section of the executable file, it will not be changed by any untrusted function. This ensures that the contents of the *symbol table* can be trusted. Second, the current instruction address is stored in the instruction pointer, which is a processor register that cannot be modified by the user level program.

With the *symbol table*, *StackVault* can reliably figure out which function invokes the stack protection system call during the runtime. Any stack protection system call that is invoked within an untrusted function will be considered as an illegal system call.

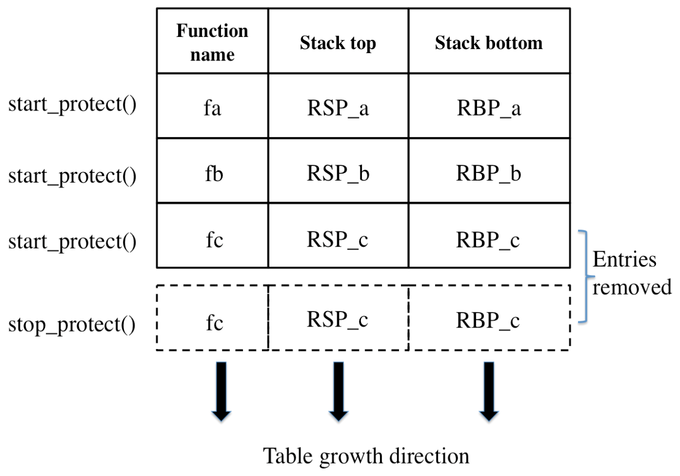


Figure 50: The protection table

Invocation correctness verification. *StackVault* checks the correctness of the invocation of stack protection system calls by making sure that *start_protect()* and *end_protect()* are invoked in pairs by the same function. For example, when *f1()* calls *stop_protect()*, *StackVault* check whether the latest operation is a *start_protect()* from *f1()* with the same RBP and RSP values. If not, this *stop_protect()* is illegal. This is because the *stop_protect()* must restore the same stack memory that has been most recently protected by the same function.

The purpose of the *protection table* is to facilitate this checking process. As shown in Figure 50, this table is maintained in a LIFO manner. When the *start_protect()* is invoked, a new entry is appended to the end of the table. The entry includes the name of the function that invokes *start_protect()*, and the value of RSP and RBP. Since each function has its own stack, the range specified by current RSP and RBP should not be overlapped with any other ranges existing in the table. Whenever a *stop_protect()* is matched with the latest *start_protect()*, the latest *start_protect()* entry is removed from the table.

6.3.4 Handling Complex Call Graphs

We have discussed the case of sensitive function having a direct nested call relationship with an untrusted function with example shown in Listing 6.3. However, when a sensitive function has an indirect nested call relationship with an untrusted function, things get more complicated. This is because when an untrusted function is not directly invoked by a sensitive function, if the *start_protect()* and *stop_protect()* APIs are still inserted by wrapping around the untrusted function, then it only prevents the untrusted function from accessing the stack of its direct caller, but the sensitive data on its indirect caller‘s stack is still under the risk of leakage. Thus, *StackVault* should ensure that the correct locations are identified to place the right pair of system calls in the executable code.

Figure 51 illustrate this problem and how it is solved in *StackVault*. An example of a nested call graph is given, in which there exists sensitive functions, normal functions, and an untrusted function. The untrusted function is indirectly invoked by the sensitive functions. Concretely, the sensitive data is allocated on the stack of *sensitive_func_1()* and *sensitive_func_2()*. If the *start_protect()* and *stop_protect()* are inserted before and after *untrusted_func_1()*, it only prevents the *untrusted_func_1()* from accessing the stack of *normal_func_3()*, but the sensitive stack data can still be accessed by *untrusted_func_1()*. Therefore, in order to correctly protect the sensitive data, the stack protection APIs need to be inserted in the places described in Figure 51. Specifically, if an untrusted function is indirectly invoked by a sensitive function, *StackVault* identifies the function which not only being directly invoked by the sensitive function, but also invokes the untrusted function, such as the *normal_func_1()* and *normal_func_3()* in Figure 51. Then the stack protection APIs will be inserted before and after these functions. Due to the space limit, we omit the detailed rules used in *StackVault* for correct placement of its system calls in the complex indirect nested call relationship from sensitive functions to untrusted functions.

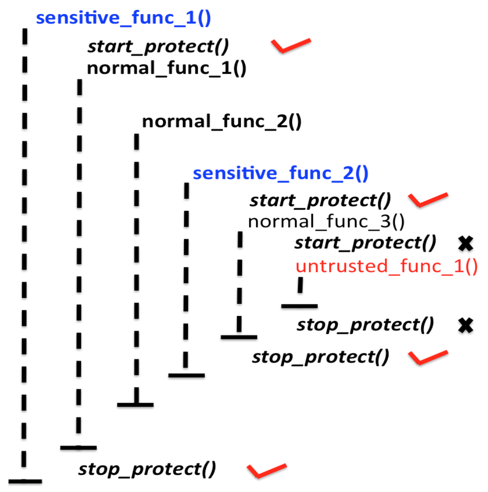


Figure 51: Insert stack protection APIs in indirect nested call scenario.

6.3.5 Other Design Choices

Resolving parameters. When a sensitive function *f1()* calls an untrusted function *f2()*, *f2()* can take parameters which locate on the stack of *f1()*. In this case, although *f2()* is not allowed to touch the other areas of *f1()*'s stack, it should be able to access its parameters. This requires that the *start_protect()* system call should be able to automatically figure out which part of *f1()*'s stack contains the data that will be accessed by *f2()* as parameters, and then keep those areas accessible by *f2()* after stack protection. This is achieved by analyzing the DWARF info, which can be acquired through the executable file.

Invocation by assembly. Although *start_protect()* and *stop_protect()* are designed as system calls, attentions need to be paid on the correct way to invoke them. In *StackVault*, we invoke *start_protect()* and *end_protect()* in assembly language using the system call number instead the name. The reason is as following.

As discussed earlier, whenever a *start_protect()* or *stop_protect()* has been invoked, *StackVault* refers to the instruction pointer to figure out what function has issued the invocation, so that any illegal invocation from an untrusted function can be prevented. However, when the system calls are invoked by their names, the instruction pointer that *StackVault* obtains within the system calls will point to the same address, no matter which function has invoked the system call. This is because the name of the system call is a wrapper function and the real system call is invoked inside this wrapper. Thus, if the system call is invoked by its name, the instruction pointer will point to the wrapper function, no matter where the system call is invoked.

Final Remarks. In order to protect sensitive data on the stack, in the current implementation of *StackVault*, the copy and restore approach is used, which incurs negligible performance overhead as shown in our experimental evaluation. Specifically, *StackVault* first copies the stack data, which needs to be protected, from the user level memory to the kernel buffer. Then, it clears the corresponding data in the original stack. The stack data in the kernel buffer cannot be accessed by any user level untrusted function. To restore a stack, *StackVault* puts the data back from the kernel buffer to the user level stack, and then clears and releases the kernel buffer.

Other than copying the data between the user level stacks and the kernel buffers, an in-place encryption/decryption approach will reduce the data copy, and only the keys need to be stored in the kernel buffer. However, frequent encryption and decryption could result in higher CPU overhead than simply copying the data. We are interested in providing the encryption as the option in the future.

6.4 Evaluation

In this section, we evaluate the effectiveness of *StackVault* using four widely used software packages: *Minizip*, *Curl*, *OpenSSH*, and *Netperf*.

6.4.1 Experiments Setup

All the experiments are conducted on a Softlayer [24] machine with 8 3.5 GHz Intel Xeon-Haswell CPUs, 32GB memory, and 4TB disk storage. Ubuntu 14.04-64 is used as the

Table 7: Open source software packages used in the evaluation(# represents the number of functions in each application)

Application	**Executable file**	**Category**	**LoC(k)**	**Stack Size(bytes)**	**#**
Minizip	minizip	File compression tool	7	8∼67128	48
Curl	curl	Http client	177	16∼4176	102
OpenSSH	ssh	Remote login tool	130	16∼65664	1127
Netperf	Netperf	Networking benchmark	58	16∼24640	180

operating system, and the kernel version is 4.1.0. The following four applications are used in the experiments:

- **Minizip.** It allows to deflate compressed files and to create gzip (.gz) files.
- **Curl.** It is used in command lines or scripts to transfer data. It is the internet transfer backbone for thousands of software applications affecting billions of humans daily.
- **OpenSSH.** It is the premier connectivity tool for remote login with the SSH protocol. It encrypts all traffic to eliminate eavesdropping, connection hijacking, and other attacks.
- **Netperf.** It is a benchmark that can be used to measure the perofmrnace of many different types of networking.

By default, all these applications are compiled by GCC. Since *StackVault* is based on LLVM [86], we modified the related Makefiles in each application to have them compiled with LLVM.

Table 7 summarizes the characteristics of the applications used in our experiments. Each application includes multiple functions. The number of functions each application has is displayed in the last column of the table. We can see that among these four applications, OpenSSH contains the largest number of functions while Minizip has the least number of functions. The "Stack Size" column represents the range of the stack sizes for all the functions in a given application. Consider *Minizip*, the smallest stack size of a function is 8 bytes, while the biggest stack size of a function is 67128 bytes.

Table 8 lists the stack size distribution in five categories of percentile. It is obvious to see that no matter for which application, functions with smaller stack sizes are much more popular than those with larger stack sizes. For *Minizip*, although the largest function stack size is 67,128 bytes, 79% functions have the stack size smaller than 100 bytes and the stack sizes of 98% of the functions are less than 616 bytes. This shows that most of the

Table 8: Stack size percentile per application(byte)

	20%	**40%**	**60%**	**80%**	**100%**
Minizip	8	8	24	104	67128
Curl	16	48	80	176	4176
OpenSSH	32	48	64	144	65664
Netperf	32	40	56	152	24640

Table 9: Functions from each application protected by *StackVault*

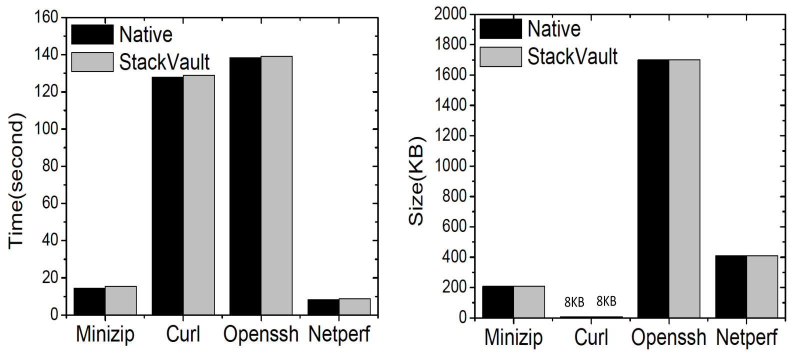
Application	**Large stack function**	**File**	**Stack size (bytes)**	**Small stack function**	**File**	**Stack size (bytes)**
Minizip	zipOpen4	zip.c	67128	fcrypt_end	fileenc.c	8
	derive_key	pwd2key.c	616	hmac_sha_key	hmac.c	8
	filetime	minizip.c	424	hmac_sha_data	hmac.c	8
Curl	my_get_line	tool_parsecfg.c	4176	slist_wc_free_all	slist_wc.c	16
	operate_do	tool_operate.c	1720	easysrc_init	tool_easysrc.c	16
	formparse	tool_formparse.c	904	easysrc_cleanup	tool_easysrc.c	16
OpenSSH	getrrestbyname	getrrestbyname.c	65664	sshbuf_init	sshbuf.c	16
	hostkeys_foreach	hostfile.c	16776	sshbuf_free	sshbuf.c	16
	sshkey_try_load_public	authfile.c	16496	sshbuf_len	sshbuf.c	16
Netperf	scan_cmd_line	netsh.c	16448	netlib_init	netlib.c	16
	scan_omni_args	nettest_omni.c	24640	print_omni_init	nettest_omni.c	16
	parse_direction	netsh.c	16440	format_number	netlib.c	16

functions in many applications have very small stack sizes. Thus, in addition to evaluate the performance overhead of *StackVault* against different applications, it would be interesting to evaluate the performance overhead of *StackVault* by varying the stack sizes of functions. We believe that *StackVault* incurs very low and negligible performance overhead for applications with many functions of small stack sizes. Since the size of function stack is a major factor that impact the performance overhead of *StackVault*, three functions with largest stack sizes and three functions with smallest stack sizes are chosen as the sensitive functions that get protected by *StackVault* from each application. Table 9 shows the name, stack size and the location of each function selected from the four popular software packages.

6.4.2 Performance Overhead

We first measure the performance overhead of *StackVault* with respect to the applications in terms of both compilation and execution time. Here the compilation time refers to the time spent on compiling the source code and generating the executable code. We also measure and compare the size of the executable file when the application is compiled with and without *StackVault*. The execution time refers to the time spent on running each of the applications when the applications are compiled with and without *StackVault*.

Compilation Efficiency. In this set of experiments, we compare the compilation time and the executable file size of each application with and without *StackVault*. With *StackVault*, the selected functions shown in Table 9 are protected with the system calls for stack protection and stack clearance. Figure 52 shows the measurement results in both compilation time and executable file size.



(a) Compilation time (b) Executable file size

Figure 52: Compilation performance comparison.

We make two interesting observations. First, *StackVault* introduces very tiny overhead in terms of the compliation time. The compilation time of *Minizip* increases by 7.0% while the compilation time of *OpenSSH* increases by 0.67%. Such overhead is mainly due to the fact that with *StackVault*, the compiler needs to check for each function whether it has been specified as an untrusted function by the user, and if yes, identify the location of protection and insert stack protection and stack clearance APIs with respect to each untrusted function. Among the four applications, *OpenSSH* has the largest compilation overhead sine it has the largest number of functions as shown in Table 7. Similarly, *Minizip* has the smallest number of functions, and it incurs the smallest increase in the compilation time when compared to *Minizip* without *StackVault*. Second, in this set of experiments, only 6 functions (3 large and 3 small function stack sizes) are protected with *StackVault* for all four applications. Thus, the executable file size does not change much when comparing the native case with the case where the application is compiled with *StackVault*.

It is worth to note that even the compilation time and the executable file size will further increase if more functions in an application are to be protected by *StackVault*, it is practical to say that *StackVault* deals with only those functions that has sensitive data on their stacks in the presence of untrusted third party functions, and the number of such functions are limited in many applications. Furthermore, for each sensitive function being protected, the main overhead for *StackVault* is the time spent on making the decision of where to insert the stack protection APIs in the executable file, taking into consideration of all the untrusted functions that are nested within this sensitive function. When multiple sensitive functions are nested one after another, batch based analysis can be conducted to further reduce the cost and the compilation time.

Execution Efficiency. In this set of experiments, we evaluate the overhead of *StackVault* on the application runtime. To show how *StackVault* may affect the execution efficiency of the applications, we compare the execution performance of each application for two cases: (1) *Native*, in which the application is compiled and running without *StackVault*; and (2) ***StackVault***, in which the application is compiled and run with *StackVault* enabled to protect the user-identified sensitive functions against the user-identified untrusted functions. We also measure how *StackVault* impact on the runtime of each function it protects. For each sensitive function, the overhead of *StackVault* on the application runtime is primarily the time that *StackVault* uses to create the appropriate kernel buffer for the program, to hide the sensitive stack data in the kernel buffer and then to restore the data back to the stack.

For each application, all the selected functions are involved during the evaluation measurements. Concretely, *Minizip* is executed by compressing a file with a password, *Curl* is tested by retrieving a web page using a specified configuration file, *OpenSSH* is evaluated by connecting to a specific host, and *Netperf* is measured by conducting an omni[14] test, in which requests and responses are sent between the client and the server.

The execution time of each application in each case is measured and displayed in Figure 53. We observe that for each application, its execution time in the *StackVault* case is slightly higher than that in the *Native* case. For example, the execution time of Minizip and Curl is increased by 5.5% and 4.8% respectively. It indicates that the system calls for both stack protection and stack clearance in *StackVault* incur very small performance overhead on the application runtime. We also observe that *StackVault* has not much impact on the execution

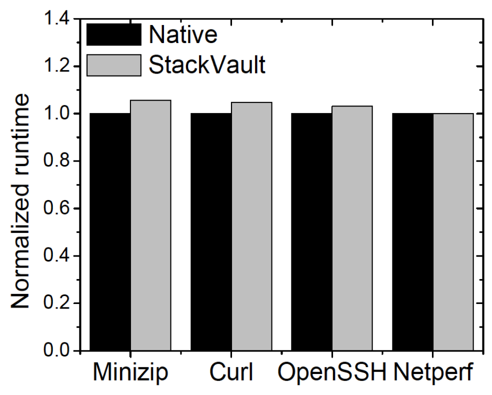


Figure 53: Application performance comparison

Table 10: Function overhead on Minizip(CPU cycle)

	Native	**StackVault**	**Overhead**
zipOpen4	180604	231968	28%
derive_key	7689k	8299k	8%
filetime	80292	92732	15%
fcrypt_end	3520	3612	3%
hmac_sha_key	1460	3692	153%
hmac_sha_data	414902	417144	1%

performance of Netperf. This is because the functions chosen to be protected in Netperf are running at the initialization and connection setup phase, which has much shorter latency than that of the bandwidth evaluation phase for Netperf.

In the next set of experiments, we measure the overhead that *StackVault* has on each function measured by *rdtsc*, which is an assembly instruction to read the time stamp counter on x86 processors. Table 10, Table 11, Table 12, and Table 13 show the per function overhead with respect to all six functions chosen for each of the four applications. We make two interesting observations: (1) For *Netperf*, the execution of all six functions is increased

Table 11: Function overhead on Curl (CPU cycle)

	Native	**StackVault**	**Overhead**
my_get_line	20196	39000	93%
operate_do	1829605k	1833081k	0.2%
formparse	57744	61724	7%
slist_wc_free_all	4060	8368	106%
easysrc_init	10488	14324	37%
easysrc_cleanup	6552	20828	218%

Table 12: Function overhead on OpenSSH (CPU cycle)

	Native	**StackVault**	**Overhead**
getrrsetbyname	345425k	357805k	4%
hostkeys_foreach	96320	139903	45%
ssh_try_load_public	5469	19719	261%
sshbuf_init	4640	18992	309%
sshbuf_free	4236	11358	168%
sshbuf_len	1472	4926	235%

Table 13: Function overhead on Netperf (CPU cycle)

	Native	**StackVault**	**Overhead**
netlib_init	138152	157608	14%
print_omni_init	119256	123516	4%
scan_omni_args	256688	325484	27%
scan_cmd_line	329928	412720	25%
parse_direction	15448	17820	15%
format_number	97564	111531	14%

in a range of 0.4% to 27%. However, for *OpenSSH*, the execution of all six functions is increased by a much larger range, from 0.4% to 309%. (2) The longer a function runs in native case, the less performance impact *StackVault* will have on its runtime. For example, the function *getrrestbyname()* runs for 345425k CPU cycles, and its runtime with *StackVault* has increased by only 4%. In contrast, *sshbuf_init()* runs for 4640 CPU cycles, and its execution time grows by 309% with *StackVault*. The reason is intuitive. The per-function overhead mainly comes from two factors: (i) The two stack protections functions provided by *StackVault* are system calls, and both of them need to be invoked in order to protect the stack of a specific function. Each invocation includes a context switch between the user space and the kernel space. (ii) The *start_protect()* function needs to allocate kernel memory buffer by *kmalloc()* to store the stack data, such kernel memory allocation also introduce overhead. Therefore, the time spent by *StackVault* on each protected function is usually similar and largely depends on the amount of sensitive data on the stack, which is relatively small and independent of the actual execution time of the function. Thus, when the execution time in the native case is short, the runtime of the function by *StackVault* will be longer in comparison. When the execution time in the native case is much longer, the increase of the runtime for the function protected by *StackVault* relatively speaking will be much smaller.

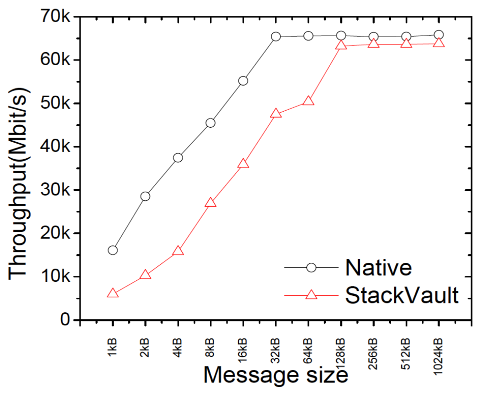


Figure 54: Netperf performance comparison under frequent invocation of *StackVault* APIs

Impact of frequent invocation. We next evaluate the impact of *StackVault* on the execution performance of an application when a frequently invoked function is under the protection of *StackVault*. Figure 54 shows the measurement results for Netperf with and without *StackVault* and the function *send_data()* is chosen since it is a frequently invoked function during the execution of Netperf. We observe that for small message size, *StackVault* does introduce more performance overhead compared to native case. However, the overhead decreases with the growth of the message size. For example, the performance overhead drops from 31.6% to 3.2% when the message size increases from 32KB to 1024KB. This is because when sending the same amount of data, the case with larger size messages will result in less frequent invocation of the *send_data()*function. In addition, we also observe that more than 95% of the overhead is brought by *start_protect()* and *stop_protect()*, while *clear_stack()* only incurs less 5%. The reason is obvious: the *clear_stack()* is a procedure instead of a system call, and it simply zeros out a specific range of memory without any additional operation.

Impact of large stack sizes. The final set of experiments is to measure the impact of stack sizes on the performance overhead of *StackVault*. An example piece of code from the libcurl website, showing how to use libcurl to upload files to a server, is used here. Since there is only a few functions in this code, it is easier to manually adjust the stack size and the effect will be more obvious. Originally, the stack size of the *uploadfile()* function is 244 bytes. We manually increase its stack size up to 4MB by allocating an array with

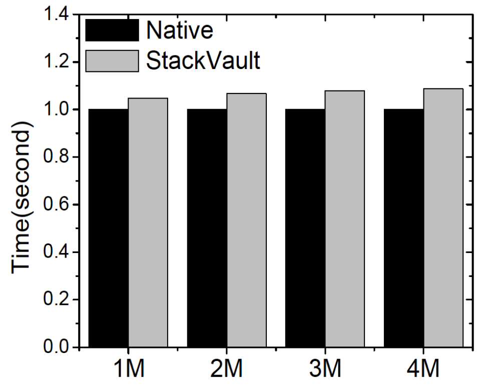


Figure 55: Protection overhead for large stack

specific size. Then, the runtime of the application is measured and compared under the *Native* scenario and the *StackVault* scenario. Figure 55 shows the overhead that *StackVault* brings when protecting large stacks. When the stack size grows from 1MB to 4MB, the performance overhead only slightly increases from 4.6% to 8.6%. considering the fact that most of the functions have a much smaller stack size varying from tens of bytes to hundreds of bytes, stack size will not become an issue for the performance of *StackVault*. Combining Figure 54 and Figure 55, it shows that *StackVault* incurrs higher overhead by protecting a highly frequently invoked function than protecting a large stack.

6.4.3 Memory Overhead

The memory overheads of *StackVault* are mainly from two factors. One is the kernel buffers used to store the stack data, the size of which is the same as the stack being protected. The other is the in-memory data structure *protection table*. The protection table is a data structure that prevents the stack protection functions from being invoked intentionally by the attackers to get the stack data that are previously protected by a legitimate user. For a specific application, the protection table is created and stored in the memory before an application starts, and then remains unchanged. We measured the size of the protection table for each application, and Table 14 shows the results in terms of the size of the executable file, its corresponding protection table size, and the ratio of protection table size over the executable file size. We observe that for all applications, the size of the protection table is very small compared to the corresponding executable file size. Among the four applications,

Table 14: Size of the in-memory protection table

Application	Executable file size	Protection table size	Overhead
Minizip	208KB	6KB	2.9%
Curl	8KB	13KB	162.5%
OpenSSH	1700KB	141KB	8.3%
Lighttpd	840KB	49KB	5.8%

OpenSSH has the largest protection table size, which is 141KB. This is consistent with our previous observation that OpenSSH has the largest number of functions. This is because for an application, the size of its protection table is proportional to the number of functions it has.

6.5 Related Work

Due to the highly predictable layout of the stack memory, stack based attacks have been existed for a long time, in which the most common one is buffer overflow [47]. For example, StackGuard [55] proposes two techniques to overcome the buffer overflow vulnerability. One is putting a canary word right besides each return address on the stack, so that the modification of the return address can be detected by checking whether the canary word has been changed. This idea has also been incorporated in the GCC [7] compiler. The other technique takes the advantage of the debug registers to monitor the stack memory that stores the return address, and triggers an exception once any return address has been rewritten. CRED [111] introduces a C range error detector, which allows program access out-of-bounds addresses that do not result in buffer overflows. Other stack protection approaches such as ASLR [116] and StackArmor [49] use randomization to make it difficult for the attackers to guess where the target stack frame is. The shadow stack [118, 57] was invented to protect return address on the stack from tampering. In this scheme, a shadow stack is maintained in parallel with the original stack, which is used to ensure the integrity of the address. Control-Flow Integrity (CFI) based approaches [37, 56, 127, 36] are also designed to protect the stack-based buffer overflows. Such approaches first construct a Control-Flow Graph (CFG) using source code analysis, binary analysis, or executing profiling. Then, the software execution will be dictated to follow the CFG, so that a compromised execution path caused by buffer overflow will be prevented. All the above-mentioned techniques are focusing on eliminating the buffer overflow stack, but the other sensitive data on the stack also needs to be appropriately protected to prevent data leakage.

Many researchers also focus on eliminating the in-memory sensitive data leakage. For instance, Shreds [51] protects the sensitive information in private memory by using the memory domain features in ARM CPU, and [122] explores the trust issues in multithreaded applications such as MemCached. [74] finds that DRAM can retain its data for several seconds after it is powered off and removed from the motherboard, and briefly discusses several solutions to these attacks, such as changing the architecture of the DRAM to make it lose state more quickly. SWPIE [72, 71] takes the advantages of static analysis to erased the sensitive data at the earliest time. [52] presents a secure deallocation strategy to reduce the life cycle of the sensitive information in memory. Vanish [70] aims at creating self-destruct data that can automatically vanish when it is no longer useful. All these efforts effectively reduce the lifetime of the sensitive information in memory so it is less possible to be leaked. However, as discussed in this chapter, sensitive data can also be leaked before its lifetime finishes if some function in a program is compromised. Data leakages also happen via uninitialized read. In this case, an attacker can get the stack data via reading uninitialized stack variables. [98] and [93] are two recent efforts that solve this problem by explicitly initialized each local variable after it is allocated on the stack.

6.6 Conclusions

Untrusted third party functions are becoming a significant threat for stack data leakage. In this chapter, we designed and implemented *StackVault*, which is a highly reliable and transparent tool to protect the sensitive data on the function stack. Taking the advantage of the OS kernel, *StackVault* prevents the sensitive data on the stack from being illegally accessed by any other untrusted functions in the same process. We evaluated the effectiveness of *StackVault* using four widely accepted applications. Results show that *StackVault* occurs fairly low overhead on the runtime of the applications in most cases. In the future, *StackVault* can be extended to several directions. On one hand, we are considering not only protect the data on the stack, but also the data on the heap. On the other hand, we plan to investigate how to prevent memory data leakage among different virtual machines running on the same host.

Chapter VII

CONCLUSIONS AND FUTURE WORK

With the increasing demands for fast big data processing, memory is becoming one of the critical resources to improve the performance of the applications in the Cloud. Large size memory machines and platforms are not uncommon nowadays, and many applications can benefit from using large memory to avoid expensive disk I/O operations. However, how to efficiently and securely use such large memory is still an open issue. Maximizing the memory utilization while maintaining QoS for each individual application and guaranteeing the in-memory data security is still challenging to existing Cloud providers.

7.1 Summary

This dissertation makes four unique contributions. First, *MemPipe* is introduced as a dynamic shared memory framework for high performance communication and data transfer among co-located VMs in virtualized cloud. *MemPipe* employs an inter-VM shared memory pipe to enable high throughput data delivery for both TCP and UDP workloads among co-located VMs. Instead of static shared memory allocation, *MemPipe* manages its shared memory pipes through a demand driven and proportional memory allocation mechanism, which can dynamically enlarge or shrink the shared memory pipes based on the demand of the workloads from the VMs. Furthermore, *MemPipe* employs a number of optimizations such as *time-window based streaming partitions* and *socket buffer redirection* to further improve the performance of co-located inter-VM communication.

Second, we have presented the design of *MemFlex*, a highly efficient shared memory swapper. *MemFlex* makes three original contributions. First, *MemFlex* can effectively utilize host idle memory by redirecting the VM swapping traffic to the host-guest shared memory swap area. Second, the hybrid memory swap-out model in *MemFlex* promotes to use the fast shared memory swap partition as the primary swap area whenever possible, and smoothly transits to the conventional disk-based VM swapping scheme on demand. Third but not the least, *MemFlex* proactive swap-in optimization offers just-in-time performance recovery by replacing costly page faults with an efficient swap-in implementation.

In addition, a low cost VM memory balancer, iBalloon, with high accuracy and transparency is proposed. No modification is required for VMs or the hypervisor to deploy *iBalloon*, which makes it more acceptable in real Cloud environment. *iBalloon* runs a light-weighted monitoring daemon in each VM, which gathers the information about memory utilization of that VM. At the same time, a balancer daemon is running in the host to collect information reported by the monitor, and automatically makes the decision about how to balance the memory around VMs. The balancer finally talks to the Balloon driver in the host machine to actually move the memory around.

Last but not the least, a kernel-backed system-level facility, *StackVault*, is designed to eliminate sensitive stack data leakage. *StackVault* enforces three types of stack protection operations to protect the sensitive stack data by preventing an untrusted function from illegally accessing the stack of another function in the same process. Through placement and enforcement of such operations, *StackVault* moves the sensitive stack data into an OS kernel buffer prior to the execution of an untrusted function, so that there is no way for such data to be touched by any untrusted function. Such protection also ensures that all data required for the execution of the untrusted function is kept on the stack. The stack data is restored immediately after the untrusted function returns, and the stack is cleared for every sensitive function upon its return, in order to eliminate any leakage of stack data after its completion.

7.2 Future Work

There are many interesting open research problems for in-memory computing area. First of all, the Cloud platforms need an operating system (Cloud OS) to automatically manage the memory resources from different physical machines as a whole. The Cloud OS should provide an uniformed abstraction of the underlying memory resources to the VMs, which enable a VM to expand its memory across multiple physical machines. This mechanism can further increase the memory utilization of the whole Cloud platform. However, it faces multiple challenges, For example, different access latencies between local memory and remote memory need to be considered so as to minimize the VM performance overhead. Also, as the memory data of a single VM or a single application is now widely spread among the Cloud, being able to quickly recover from the machine failure is becoming essential. Another thread of future research will be making the data secure in virtualized environment. Memory data leakage can happen not only within a VM, but also among multiple VMs when they are running on the host. Therefore, it is critical to effectively and efficiently track the memory data, which is generated in the VM and the travels out side of the VM while accessing the hardware device, and clears the memory as early as possible. This ensures that memory data in the VM will not be leaked.

REFERENCES

[1] "Advances in Memory Management in a Virtual Environment," *https://oss.oracle.com/projects/tmem/dist/documentation/presentations/MemMgmtVirtEnv-LPC2010-Final.pdf.*

[2] "Auto Ballooning," *http://www.linux-kvm.org/page/Projects/auto-ballooning.*

[3] "Compromising US Banks with Third-party Code.," *https://blog.gaborszathmari.me/2016/02/11/compromising-us-banks-third-party-code/.*

[4] "Dacapo," *http://www.dacapobench.org.*

[5] "DPDK Notes Experimental," *https://github.com/wanduow/libtrace/wiki/DPDK-Notes—Experimental.*

[6] "Frontswap," *https://www.kernel.org/doc/Documentation/vm/frontswap.txt.*

[7] "GCC, the GNU Compiler Collection," *https://gcc.gnu.org/.*

[8] "Himeno," *http://accc.riken.jp/2444.htm.*

[9] "Httperf," *http://www.hpl.hp.com/research/linux/httperf.*

[10] "Intel-DPDK," *Intel Corporation. intel data plane development kit: Getting started guide. 2013.*

[11] "Intel MPI Benchmarks," *https://software.intel.com/en-us/articles/intel-mpi-benchmarks.*

[12] "Memcached," *https://memcached.org/.*

[13] "Memtier benchmark," *https://github.com/RedisLabs.*

[14] "Netperf," *https://launchpad.net/ubuntu/precise/+package/netpipe-mpich.*

[15] "Netpipe-mpich," *http://www.netperf.org/netperf/.*

[16] "OSU MPI Benchmarks," *http://mvapich.cse.ohio-state.edu/benchmarks/.*

[17] "OSU MPI benchmarks," *http://mvapich.cse.ohio-state.edu/benchmarks/.*

[18] "Overview of Single Root I/O Virtualization (SR-IOV)," *http://msdn.microsoft.com/en-us/library/windows/hardware/hh440148%28v=vs.85%29.aspx.*

[19] "Perf," *https://perf.wiki.kernel.org/index.php/Tutorial.*

[20] "Protecting Sensitive Information in Memory.," *https://www.controlscan.com/blog/protecting-sensitive-info-memory/.*

[21] "QEMU," *http://wiki.qemu.org/.*

[22] "Redis," *http://redis.io.*

[23] "Sftp," *http://en.wikipedia.org/wiki/SFTP.*

[24] "Softlayer," *http://www.softlayer.com.*

[25] "SPECjvm2008," *http://www.spec.org/jvm2008.*

[26] "Stack Shield.," *http://www.angelfire.com/sk/stackshield/.*

[27] "Sysbench," *https://wiki.mikejung.biz/Sysbench.*

[28] "Third-party libraries are one of the most insecure parts of an application.," *https://techbeacon.com/third-party-libraries-are-one-most-insecure-parts-application.*

[29] "Understanding Memory Resource Management in VMware vSphere 5.0," *VMware Technical White Paper, August 2011.*

[30] "Vsftp," *https://security.appspot.com/vsftpd.html.*

[31] "Wget," *https://www.gnu.org/software/wget/.*

[32] "Cve-2012-1889: Vulnerability in microsoft xml core services could allow remote code execution.," 2012.

[33] "Cve-2014-0160: The heartbleed vulnerability.," 2014.

[34] "Cve-2015-0240: Unexpected code execution in smbd," 2015.

[35] "Cve-2015-7547: The buffer overflow vulnerability in a domain-name lookup function.," 2015.

[36] Abadi, M., Budiu, M., Erlingsson, U., and Ligatti, J., "Control-flow integrity," in *Proceedings of the 12th ACM conference on Computer and communications security*, pp. 340–353, ACM, 2005.

[37] Abadi, M., Budiu, M., Erlingsson, Ú., and Ligatti, J., "Control-flow integrity principles, implementations, and applications," *ACM Transactions on Information and System Security (TISSEC)*, vol. 13, no. 1, p. 4, 2009.

[38] Ahn, J., Kim, C., Han, J., Choi, Y.-r., and Huh, J., "Dynamic virtual machine scheduling in clouds for architectural shared resources," in *Presented as part of the*, 2012.

[39] Amit, N., Tsafrir, D., and Schuster, A., "Vswapper: A memory swapper for virtualized environments," in *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*, pp. 349–366, ACM, 2014.

[40] Amza, C., Cox, A. L., Dwarkadas, S., Keleher, P., Lu, H., Rajamony, R., Yu, W., and Zwaenepoel, W., "Treadmarks: Shared memory computing on networks of workstations," *Computer*, vol. 29, no. 2, pp. 18–28, 1996.

[41] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A., "Xen and the art of virtualization," *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5, pp. 164–177, 2003.

[42] Birke, R., Chen, L. Y., and Smirni, E., "Data centers in the wild: A large performance study," *Technical Repo*, no. Z1204-002, 2012.

[43] Blackburn, S. M., Garner, R., Hoffmann, C., Khang, A. M., McKinley, K. S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S. Z., and others, "The dacapo benchmarks: Java benchmarking development and analysis," in *ACM Sigplan Notices*, vol. 41, pp. 169–190, ACM, 2006.

[44] Burtsev, A., Srinivasan, K., Radhakrishnan, P., Voruganti, K., and Goodson, G. R., "Fido: Fast inter-virtual-machine communication for enterprise appliances.," in *USENIX Annual technical conference*, San Diego, CA, 2009.

[45] Chang, K. K.-W., Lee, D., Chishti, Z., Alameldeen, A. R., Wilkerson, C., Kim, Y., and Mutlu, O., "Improving dram performance by parallelizing refreshes with accesses," in *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pp. 356–367, IEEE, 2014.

[46] Chapman, M. and Heiser, G., "vnuma: A virtual shared-memory multiprocessor.," in *USENIX Annual Technical Conference*, 2009.

[47] Chen, H., Mao, Y., Wang, X., Zhou, D., Zeldovich, N., and Kaashoek, M. F., "Linux kernel vulnerabilities: State-of-the-art defenses and open problems," in *Proceedings of the Second Asia-Pacific Workshop on Systems*, p. 5, ACM, 2011.

[48] Chen, P. M. and Noble, B. D., "When virtual is better than real [operating system relocation to virtual machines]," in *Hot Topics in Operating Systems, 2001. Proceedings of the Eighth Workshop on*, pp. 133–138, IEEE, 2001.

[49] Chen, X., Slowinska, A., Andriesse, D., Bos, H., and Giuffrida, C., "Stackarmor: Comprehensive protection from stack-based memory error vulnerabilities for binaries.," in *NDSS*, 2015.

[50] Chen, Y., Ganapathi, A. S., Griffith, R., and Katz, R. H., "Analysis and lessons from a publicly available google cluster trace," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2010-95*, vol. 94, 2010.

[51] Chen, Y., Reymondjohnson, S., Sun, Z., and Lu, L., "Shreds: Fine-grained execution units with private memory," 2016.

[52] Chow, J., Pfaff, B., Garfinkel, T., and Rosenblum, M., "Shredding your garbage: Reducing data lifetime through secure deallocation.," in *USENIX Security*, pp. 22–22, 2005.

[53] Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R., "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM symposium on Cloud computing*, pp. 143–154, ACM, 2010.

[54] Costan, V. and Devadas, S., "Intel sgx explained.," *IACR Cryptology ePrint Archive*, vol. 2016, p. 86, 2016.

[55] Cowan, C., Pu, C., Maier, D., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q., and Hinton, H., "Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks.," in *Usenix Security*, vol. 98, pp. 63–78, 1998.

[56] Criswell, J., Dautenhahn, N., and Kcofi, V., "Complete control-flow integrity for commodity operating system kernels," in *2014 IEEE Symposium on Security and Privacy (SP)*, pp. 292–307.

[57] Dang, T. H., Maniatis, P., and Wagner, D., "The performance cost of shadow stacks and stack canaries," in *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, pp. 555–566, ACM, 2015.

[58] David, H., Fallin, C., Gorbatov, E., Hanebutte, U. R., and Mutlu, O., "Memory power management via dynamic voltage/frequency scaling," in *Proceedings of the 8th ACM international conference on Autonomic computing*, pp. 31–40, ACM, 2011.

[59] Dean, J. and Barroso, L. A., "The tail at scale," *Communications of the ACM*, vol. 56, no. 2, pp. 74–80, 2013.

[60] Dean, J. and Ghemawat, S., "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[61] Dean, J. and Ghemawat, S., "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[62] Deng, Q., Meisner, D., Ramos, L., Wenisch, T. F., and Bianchini, R., "Memscale: active low-power modes for main memory," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 1, pp. 225–238, 2011.

[63] Deshpande, U., Wang, B., Haque, S., Hines, M., and Gopalan, K., "Memx: Virtualization of cluster-wide memory," in *Parallel Processing (ICPP), 2010 39th International Conference on*, pp. 663–672, IEEE, 2010.

[64] Diakhaté, F., Perache, M., Namyst, R., and Jourdren, H., "Efficient shared memory message passing for inter-vm communications," in *Euro-Par 2008 Workshops-Parallel Processing*, pp. 53–62, Springer, 2009.

[65] Eiraku, H., Shinjo, Y., Pu, C., Koh, Y., and Kato, K., "Fast networking with socket-outsourcing in hosted virtual machine environments," in *Proceedings of the 2009 ACM symposium on Applied Computing*, pp. 310–317, ACM, 2009.

[66] Feeley, M. J., Morgan, W. E., Pighin, E., Karlin, A. R., Levy, H. M., and Thekkath, C. A., *Implementing global memory management in a workstation cluster*, vol. 29. ACM, 1995.

[67] Frantzen, M. and Shuey, M., "Stackghost: Hardware facilitated stack protection.," in *USENIX Security Symposium*, vol. 112, 2001.

[68] Gamsa, B., Krieger, O., Appavoo, J., and Stumm, M., "Tornado: Maximizing locality and concurrency in a shared memory multiprocessor operating system," in *OSDI*, vol. 99, pp. 87–100, 1999.

[69] Garg, N., *Apache Kafka.* Packt Publishing Ltd, 2013.

[70] Geambasu, R., Kohno, T., Levy, A. A., and Levy, H. M., "Vanish: Increasing data privacy with self-destructing data.," in *USENIX Security Symposium*, pp. 299–316, 2009.

[71] Gondi, K., Bisht, P., Venkatachari, P., Sistla, A. P., and Venkatakrishnan, V., "Swipe: eager erasure of sensitive data in large scale systems software," in *Proceedings of the second ACM conference on Data and Application Security and Privacy*, pp. 295–306, ACM, 2012.

[72] Gondi, K., Sistla, A. P., and Venkatakrishnan, V., "Minimizing lifetime of sensitive data in concurrent programs," in *Proceedings of the 4th ACM conference on Data and application security and privacy*, pp. 171–174, ACM, 2014.

[73] Gupta, D., Lee, S., Vrable, M., Savage, S., Snoeren, A. C., Varghese, G., Voelker, G. M., and Vahdat, A., "Difference engine: Harnessing memory redundancy in virtual machines," *Communications of the ACM*, vol. 53, no. 10, pp. 85–93, 2010.

[74] Halderman, J. A., Schoen, S. D., Heninger, N., Clarkson, W., Paul, W., Calandrino, J. A., Feldman, A. J., Appelbaum, J., and Felten, E. W., "Lest we remember: cold-boot attacks on encryption keys," *Communications of the ACM*, vol. 52, no. 5, pp. 91–98, 2009.

[75] Hines, M. R., Gordon, A., Silva, M., Da Silva, D., Ryu, K. D., and Ben-Yehuda, M., "Applications know best: Performance-driven memory overcommit with ginkgo," in *Cloud Computing Technology and Science (CloudCom), 2011 IEEE Third International Conference on*, pp. 130–137, IEEE, 2011.

[76] Hoffman, S., *Apache Flume: Distributed Log Collection for Hadoop.* Packt Publishing Ltd, 2013.

[77] Huang, W., Koop, M. J., Gao, Q., and Panda, D. K., "Virtual machine aware communication libraries for high performance computing," in *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, p. 9, ACM, 2007.

[78] Hwang, J., Ramakrishnan, K., and Wood, T., "Netvm: high performance and flexible networking using virtualization on commodity platforms," in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14). Seattle, WA: USENIX Association*, pp. 445–458, 2014.

[79] Johnson, K. L., Kaashoek, M. F., and Wallach, D. A., *CRL: High-performance all-software distributed shared memory*, vol. 29. ACM, 1995.

[80] Jones, S. T., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H., "Geiger: monitoring the buffer cache in a virtual machine environment," in *ACM SIGOPS Operating Systems Review*, vol. 40, pp. 14–24, ACM, 2006.

[81] Kim, K., Kim, C., Jung, S.-I., Shin, H.-S., and Kim, J.-S., "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*, pp. 11–20, ACM, 2008.

[82] Kim, Y., Seshadri, V., Lee, D., Liu, J., and Mutlu, O., “A case for exploiting subarray-level parallelism (salp) in dram,” in *Computer Architecture (ISCA), 2012 39th Annual International Symposium on*, pp. 368–379, IEEE, 2012.

[83] Kistler, M. and Alvisi, L., “Improving the performance of software distributed shared memory with speculation,” *Parallel and Distributed Systems, IEEE Transactions on*, vol. 16, no. 9, pp. 885–896, 2005.

[84] Kivity, A., Kamay, Y., Laor, D., Lublin, U., and Liguori, A., “kvm: the linux virtual machine monitor,” in *Proceedings of the Linux Symposium*, vol. 1, pp. 225–230, 2007.

[85] Lange, J. R. and Dinda, P., “Symcall: Symbiotic virtualization through vmm-to-guest upcalls,” in *ACM SIGPLAN Notices*, vol. 46, pp. 193–204, ACM, 2011.

[86] Lattner, C. and Adve, V., “Llvm: A compilation framework for lifelong program analysis & transformation,” in *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, p. 75, IEEE Computer Society, 2004.

[87] Lee, D., Kim, Y., Pekhimenko, G., Khan, S., Seshadri, V., Chang, K., and Mutlu, O., “Adaptive-latency dram: Optimizing dram timing for the common-case,” in *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, pp. 489–501, IEEE, 2015.

[88] Lee, D., Kim, Y., Seshadri, V., Liu, J., Subramanian, L., and Mutlu, O., “Tiered-latency dram: A low latency and low cost dram architecture,” in *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, pp. 615–626, IEEE, 2013.

[89] Li, X., Wu, J., Tang, S., and Lu, S., “Let’s stay together: Towards traffic aware virtual machine placement in data centers,” in *INFOCOM, 2014 Proceedings IEEE*, pp. 1842–1850, IEEE, 2014.

[90] Lim, K., Chang, J., Mudge, T., Ranganathan, P., Reinhardt, S. K., and Wenisch, T. F., “Disaggregated memory for expansion and sharing in blade servers,” in *ACM SIGARCH Computer Architecture News*, vol. 37, pp. 267–278, ACM, 2009.

[91] Lin, C.-C., Liu, P., and Wu, J.-J., “Energy-aware virtual machine dynamic provision and scheduling for cloud computing,” in *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pp. 736–737, IEEE, 2011.

[92] Liu, H., Jin, H., Liao, X., Deng, W., He, B., and Xu, C.-z., “Hotplug or ballooning: A comparative study on dynamic memory management techniques for virtual machines,” 2013.

[93] Lu, K., Song, C., Kim, T., and Lee, W., “Unisan: Proactive kernel memory initialization to eliminate data leakages,” in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pp. 920–932, ACM, 2016.

[94] Lu, P. and Shen, K., “Virtual machine memory access tracing with hypervisor exclusive cache.,” in *Usenix Annual Technical Conference*, pp. 29–43, 2007.

[95] Macdonell, A. C., *Shared-memory optimizations for virtual machines.* PhD thesis, University of Alberta, 2011.

[96] Magenheimer, D., Mason, C., McCracken, D., and Hackel, K., "Transcendent memory and linux," in *Proceedings of the Linux Symposium*, pp. 191–200, 2009.

[97] Meng, X., Pappas, V., and Zhang, L., "Improving the scalability of data center networks with traffic-aware virtual machine placement," in *INFOCOM, 2010 Proceedings IEEE*, pp. 1–9, IEEE, 2010.

[98] Milburn, A., Bos, H., and Giuffrida, C., "Safeinit: Comprehensive and practical mitigation of uninitialized read vulnerabilities," 2017.

[99] Mohebbi, H. R., Kashefi, O., and Sharifi, M., "Zivm: A zero-copy inter-vm communication mechanism for cloud computing," *Computer and Information Science*, vol. 4, no. 6, p. p18, 2011.

[100] Moltó, G., Caballer, M., Romero, E., and de Alfonso, C., "Elastic memory management of virtualized infrastructures for applications with dynamic memory requirements," *Procedia Computer Science*, vol. 18, pp. 159–168, 2013.

[101] Mutlu, O. and Subramanian, L., "Research problems and opportunities in memory systems," *Supercomputing Frontiers and Innovations*, vol. 1, no. 3, p. 19, 2014.

[102] Omondi, A. and Sedukhin, S., *Advances in Computer Systems Architecture: 8th Asia-Pacific Conference, ACSAC 2003, Aizu-Wakamatsu, Japan, September 23-26, 2003, Proceedings*, vol. 2823. Springer Science & Business Media, 2003.

[103] Pekhimenko, G., Huberty, T., Cai, R., Mutlu, O., Gibbons, P. B., Kozuch, M. A., and Mowry, T. C., "Exploiting compressed block size as an indicator of future reuse," in *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, pp. 51–63, IEEE, 2015.

[104] Pekhimenko, G., Mowry, T. C., and Mutlu, O., "Linearly compressed pages: A main memory compression framework with low complexity and low latency," in *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, pp. 489–490, ACM, 2012.

[105] Pekhimenko, G., Seshadri, V., Mutlu, O., Mowry, T. C., Gibbons, P. B., and Kozuch, M. A., "Base-delta-immediate compression: A practical data compression mechanism for on-chip caches," *PACT-21*, 2012.

[106] Radhakrishnan, P. and Srinivasan, K., "Mmnet: An efficient inter-vm communication mechanism," *Proc. of Xen Summit. Boston*, 2008.

[107] Rao, J., Wang, K., Zhou, X., and Xu, C.-Z., "Optimizing virtual machine scheduling in numa multicore systems," in *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, pp. 306–317, IEEE, 2013.

[108] Ren, Y., Liu, L., Zhang, Q., Wu, Q., Wu, J., Kong, J., Guan, J., and Dai, H., "Residency-aware virtual machine communication optimization: Design choices

and techniques," in *2013 IEEE 6th International Conference on Cloud Computing (CLOUD)*, pp. 823–830, IEEE, 2013.

[109] Rosenblum, M., "Vmware's virtual platform," in *Proceedings of hot chips*, vol. 1999, pp. 185–196, 1999.

[110] Russell, R., "virtio: towards a de-facto standard for virtual i/o devices," *ACM SIGOPS Operating Systems Review*, vol. 42, no. 5, pp. 95–103, 2008.

[111] Ruwase, O. and Lam, M. S., "A practical dynamic buffer overflow detector.," in *NDSS*, vol. 4, pp. 159–169, 2004.

[112] Salomie, T.-I., Alonso, G., Roscoe, T., and Elphinstone, K., "Application level ballooning for efficient server consolidation," in *Proceedings of the 8th ACM European Conference on Computer Systems*, pp. 337–350, ACM, 2013.

[113] Schopp, J. H., Fraser, K., and Silbermann, M. J., "Resizing memory with balloons and hotplug," in *Proceedings of the Linux Symposium*, vol. 2, p. 313319, 2006.

[114] Schwidefsky, M., Franke, H., Mansell, R., Raj, H., Osisek, D., and Choi, J., "Collaborative memory management in hosted linux environments," in *Proceedings of the Linux Symposium*, vol. 2, 2006.

[115] Schwidefsky, M., Franke, H., Mansell, R., Raj, H., Osisek, D., and Choi, J., "Collaborative memory management in hosted linux environments," in *Proceedings of the Linux Symposium*, vol. 2, 2006.

[116] Shacham, H., Page, M., Pfaff, B., Goh, E.-J., Modadugu, N., and Boneh, D., "On the effectiveness of address-space randomization," in *Proceedings of the 11th ACM conference on Computer and communications security*, pp. 298–307, ACM, 2004.

[117] Sharma, P. and Kulkarni, P., "Singleton: system-wide page deduplication in virtual environments," in *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing*, pp. 15–26, ACM, 2012.

[118] Sinnadurai, S., Zhao, Q., and fai Wong, W., "Transparent runtime shadow stack: Protection against malicious return address modifications," 2008.

[119] Tasoulas, E., Haugerund, H., and Begnum, K., "Bayllocator: a proactive system to predict server utilization and dynamically allocate memory resources using bayesian networks and ballooning," in *Proceedings of the 26th international conference on Large Installation System Administration: strategies, tools, and techniques*, pp. 111–122, USENIX Association, 2012.

[120] Waldspurger, C. A., "Memory resource management in vmware esx server," *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, pp. 181–194, 2002.

[121] Wang, J., Wright, K.-L., and Gopalan, K., "Xenloop: a transparent high performance inter-vm network loopback," in *Proceedings of the 17th international symposium on High performance distributed computing*, pp. 109–118, ACM, 2008.

[122] Wang, J., Xiong, X., and Liu, P., "Between mutual trust and mutual distrust: practical fine-grained privilege separation in multithreaded applications," in *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pp. 361–373, 2015.

[123] Williams, D., Jamjoom, H., Liu, Y.-H., and Weatherspoon, H., "Overdriver: Handling memory overload in an oversubscribed cloud," in *ACM SIGPLAN Notices*, vol. 46, pp. 205–216, ACM, 2011.

[124] Wood, T., Tarasuk-Levin, G., Shenoy, P., Desnoyers, P., Cecchet, E., and Corner, M. D., "Memory buddies: exploiting page sharing for smart colocation in virtualized data centers," in *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pp. 31–40, ACM, 2009.

[125] Yu, J. and Narayanasamy, S., "A case for an interleaving constrained shared-memory multi-processor," in *ACM SIGARCH Computer Architecture News*, vol. 37, pp. 325–336, ACM, 2009.

[126] Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., and Stoica, I., "Spark: Cluster computing with working sets.," *HotCloud*, vol. 10, pp. 10–10, 2010.

[127] Zhang, C., Wei, T., Chen, Z., Duan, L., Szekeres, L., McCamant, S., Song, D., and Zou, W., "Practical control flow integrity and randomization for binary executables," in *Security and Privacy (SP), 2013 IEEE Symposium on*, pp. 559–573, IEEE, 2013.

[128] Zhang, P., Li, X., Chu, R., and Wang, H., "Hybridswap: A scalable and synthetic framework for guest swapping on virtualization platform," in *Computer Communications (INFOCOM), 2015 IEEE Conference on*, pp. 864–872, IEEE, 2015.

[129] Zhang, Q. and Liu, L., "Shared memory optimization in virtualized cloud," in *Cloud Computing (CLOUD), 2015 IEEE 8th International Conference on*, pp. 261–268, IEEE, 2015.

[130] Zhang, Q., Liu, L., Ren, J., Su, G., and Iyengar, A., "iBalloon: Efficient VM Memory Balancing as a Service," *2016 IEEE International Conference on Web Services (ICWS)*, pp. 33–40, 2016.

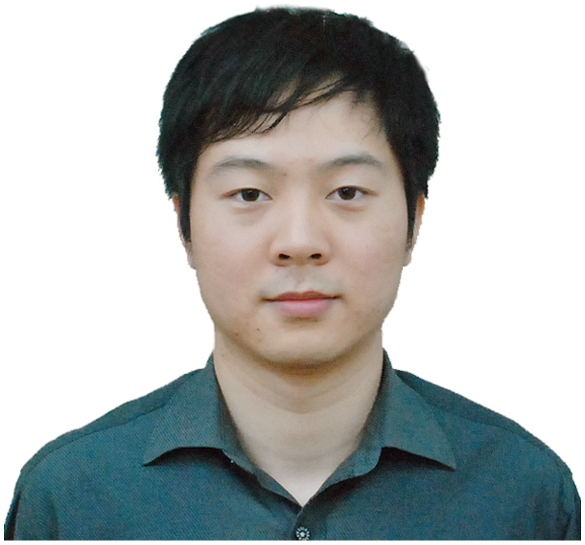
[131] Zhang, Q., Liu, L., Ren, Y., Lee, K., Tang, Y., Zhao, X., and Zhou, Y., "Residency aware inter-vm communication in virtualized cloud: Performance measurement and analysis," in *Cloud Computing (CLOUD), 2013 IEEE Sixth International Conference on*, pp. 204–211, IEEE, 2013.

[132] Zhang, X., McIntosh, S., Rohatgi, P., and Griffin, J. L., "Xensocket: A high-throughput interdomain transport for virtual machines," in *Middleware 2007*, pp. 184–203, Springer, 2007.

[133] Zhao, W., Wang, Z., and Luo, Y., "Dynamic memory balancing for virtual machines," *ACM SIGOPS Operating Systems Review*, vol. 43, no. 3, pp. 37–47, 2009.

[134] Zhou, P., Pandey, V., Sundaresan, J., Raghuraman, A., Zhou, Y., and Kumar, S., "Dynamic tracking of page miss ratio curve for memory management," in *ACM SIGOPS Operating Systems Review*, vol. 38, pp. 177–188, ACM, 2004.

VITA



Qi Zhang was born and raised in Wuhan, a beautiful city along the Changjiang river in middle China. He received both his Bachelor of Science degree and Master of Science degree from Huazhong University of Science and Technology, Wuhan, China in 2009 and 2012 respectively. Subsequently, he moved to Atlanta to pursue a Ph.D. in Computer Science at the College of Computing at Georgia Institute of Technology. As a member of the DiSL research group and CERCS at the College of Computing, he conducted research on various aspects of big data systems and computing system virtualization under the guidance of Professor Ling Liu. His research has resulted in numerous publications that have appeared in various international conferences and journals on computing systems and Cloud computing. He has also been a collaborator with the IBM T.J. Watson Research Center and IBM Almaden Research Center.