Stony Brook University

A Study of Virtualization Overheads

A Thesis presented

by

Kavita Agarwal

to

The Graduate School

in Partial Fulfillment of the

Requirements

for the Degree of

Master of Science

in

Computer Science

Stony Brook University

August 2015

Stony Brook University

The Graduate School

Kavita Agarwal

We, the thesis committee for the above candidate for the

Master of Science degree, hereby recommend

acceptance of this thesis.

Dr. Donald E. Porter, Thesis Advisor
Assistant Professor, Computer Science

Dr. Scott D. Stoller, Thesis Committee Chair
Professor, Computer Science

Dr. Mike Ferdman
Assistant Professor, Computer Science

This thesis is accepted by the Graduate School

Charles Taber
Dean of the Graduate School

Abstract of the Thesis

A Study of Virtualization Overheads

by

Kavita Agarwal

Master of Science

in

Computer Science

Stony Brook University

2015

Containers or OS-based virtualization have seen a recent resurgence in deployment. Containers have low memory footprint and start-up time but provide weaker security isolation than Virtual Machines (VMs). Incapability to load kernel modules and support multiple OS, and platform-dependence limits the functionality of containers. On the other hand, VMs or hardware-based virtualization are platform-independent and are more secure, but have higher overheads. A data centre operator chooses among these two virtualization technologies —VMs and containers—when setting up guests on cloud infrastructure. *Density* and *Latency* are two critical factors for a data centre operator because they determine the efficiency of cloud computing. Therefore, this thesis contributes updated density and latency measurements of KVM VMs and Linux Containers with a recent kernel version and best practices. This work also studies the memory footprint of KVM VMs and Linux Containers. In addition, it identifies three ways to improve the density of KVM VMs by lowering the memory footprint: improving existing memory deduplication techniques, removing unused devices emulated by QEMU, and removing unused pages from the guest address space.

To Maa, Dado, Amma, Baba, Didi, Bhai,
Teachers
and
The Divine!

Contents

List of Figures

List of Tables

List of Abbreviations

OS	Operating System
VM	Virtual Machine
KVM	Kernel Virtual Machine
LXC	Linux Container
EPT	Extended Page Table
VMM	Virtual Machine Monitor
UTS	Unix Time Sharing
PID	Process IDentifier
IPC	Inter Process Communication
API	Application Programming Interface
COW	Copy On Write
VHD	Virtual Hard Disk
CBPS	Content Based Page Sharing
ASLR	Address Space Layout Randomization
RSS	Resident Set Size
PSS	Proportional Set Size
USS	Unique Set Size
MMU	Memory Management Unit
IDT	Interrupt Descriptor Table
EOI	End Of Interrupt
SRIOV	Single Root I/O Virtualization
CRIU	Checkpoint Restore In Userspace
VMA	Virtual Memory Area
PFN	Page Frame Number
LRU	Least Recently Used

Acknowledgments

This work would not have been possible without the guidance and support of following people.

I would like to thank my advisor Prof. Donald E. Porter for giving me an opportunity to be a part of OSCAR Lab (Operating Systems Security Concurrency and Architecture Research Lab). He guided me throughout my masters and it was his continued support and belief in me which motivated me to pursue thesis along with my masters.

I would also like to thank my Thesis Committee members, Prof. Scott Stoller - Thesis Committee Chair, Prof. Mike Ferdman and my Thesis Advisor Prof. Don Porter for their time and valuable feedback for improving this work.

I would also like to acknowledge the time and efforts of Bhushan Jain, a Ph.D. student at OSCAR lab who mentored me throughout the project. He played a fundamental role in shaping this work. Sincere thanks to Chia Che and Prashant Pandey, Ph.D. students at Stony Brook University who helped me morally and technically throughout this journey. It was a great journey learning and being part of OSCAR lab.

At last, I would like to take this opportunity to thank my family members and friends who supported me in my decision of pursuing thesis and were always there for me. Special thanks to my cousin Astha Gautam for proofreading the document.

This journey was not alone and all the above players deserve to be sincerely thanked.

Chapter 1

Introduction

Cloud computing has gained importance because it makes computing cheaper. The cost of licensing software and use of hardware is reduced as the users pay a monthly/hourly fees to use the cloud which includes hardware, software, and maintenance costs. One aspect of cloud computing efficiency is the number of guests per physical machine, i.e., *density*. Higher density ensures lower cost of cloud computing because more machines can run on the same physical infrastructure, which reduces the cost incurred by an individual machine in the cloud. *Latency*, i.e., the time to provision a guest to service a request, is yet another aspect which affects efficiency of cloud computing. Lower latency helps a data centre operator to provision guests for peak load, which helps to meet the service level objectives (SLOs). Thus, for setting up a cloud infrastructure, it is important for a data centre operator to choose a guest technology (VMs or containers) that has *high density* and *low latency*.

Virtualization techniques like VMs and containers have some overheads with respect to density and latency. It is important to understand the overheads before choosing a guest technology for deploying guests in cloud infrastructure. There are a few studies that compare the overheads of VMs and containers but they are either dated in a rapidly evolving field; narrowly focused on particular application areas; or overlook important optimizations, leading to exaggerated results. For example, a tech report from IBM [23] measures 50× lower start-up time of VMs than containers. Another blog post [5] concludes 6× higher memory footprint of VMs than containers. However, as VMs and Containers have a wide range of configuration space, including common optimizations, it is easy to draw exaggerated conclusions. In the interest of adding experimental data to the scientific discourse on this topic, this thesis performs *updated latency and density measurements* of VMs and containers, considering various optimizations and latest kernel versions.

One important consideration for a cloud provider is to respond to load spikes without perceivable lag. If start-up latency is too high, a data centre operator will provision for peak load instead of the average load in order to minimize worst-case request latency. Lower start-up latency is of particular value to a cloud provider, as the provider can provision for peak load without missing response deadline, such as those specified in service level objectives (SLOs). This work analyzes the start-up latency for VMs and containers. VMs have 50× higher latency than containers. However, by using existing

optimization like *checkpoint and restore state* for VMs, the latency overhead of a VM is reduced to 6× that of containers.

Another important consideration while increasing the number of guests per physical host, i.e., *density*, is to make sure that sharing the available resources does not impact CPU and I/O performance of guests. Therefore, we measure the impact of increasing density on guest CPU and I/O performance. We observe that both VMs and containers scale equally well with respect to CPU bound workloads. However, for I/O bound workloads containers perform much better than VMs. The degradation in performance of VMs is mainly because of frequent VM exits that can be reduced by considering existing optimizations. Thus, impact of density on CPU and I/O performance of both VMs and containers is fairly close. With respect to I/O, the performance curves are similar for VMs and containers, and the first-order impact is about how I/O is multiplexed—an issue that is independent of the choice of container or VM.

Memory footprint, i.e., the amount of physical memory used by a guest determines *consolidation density*. *Lower* memory footprint of a guest ensures *higher* density i.e., more number of guests on the same physical host. This thesis also analyzes memory of VMs and containers to determine which technology has lower memory footprint. We figure out that containers have lower memory footprint as compared to VMs. This work performs an asymptotic analysis of average memory footprint of VMs and containers. This analysis is performed with and without optimization to understand the gap between VMs and containers. We observe that VMs have a 10× memory overhead compared to containers. This work suggests three solutions to reduce this cost. Improvement to existing deduplication opportunities, removal of unused devices emulated by QEMU, and removal of unused pages from the guest address space can help to reduce the memory footprint of VMs by one third, thereby ensuring higher density on same physical host.

Thus, this thesis contributes updated density and latency measurements of VMs and containers considering recent kernel and optimizations; performs memory analysis of VMs and containers; analyzes incremental cost of a VM; suggests three possible ways to reduce the incremental cost of a VM which can reduce the memory footprint of a VM and thereby reduce the gap between VMs and containers.

Portions of this work also appear in [12].

1.1 Outline

The thesis is organized as follows:

- Chapter 2 gives background details on OS-level and Hardware-based virtualization, memory types and deduplication techniques.
- Chapter 3 surveys related work.
- Chapter 4 analyzes impact of increasing the density of VMs and containers on CPU and I/O performance of guests. Both, VMs and containers perform equally well with respect to CPU bound workloads. But for I/O bound workloads, VMs

perform worse than containers. However, use of existing optimization techniques can help improve this performance gap.

- Chapter 5 analyzes start-up time of VMs and containers. Though VMs have higher start-up time than containers, the gap between them can be reduced by using existing optimizations.
- Chapter 6 analyzes memory footprint of VMs and containers; describes the usage of incremental cost of a VM, and identifies methods to reduce this cost, thereby reducing the memory footprint of VMs.

Chapter 2

Background

This Chapter describes OS-level virtualization (containers) and Hardware-based virtualization (VMs). Further, this chapter provides some background about the deduplication techniques and their application with respect to VMs and containers.

2.1 OS-Level Virtualization

Operating System level virtualization shares the same kernel as that of the host and facilitates running multiple isolated user space instances. Such instances (often known as containers, virtualization engines (VE), virtual private servers (VPS) or jails) have their own resources, root file system, and run in complete isolation from the other guests. This type of virtualization usually imposes little to no overhead because programs in virtual partitions use the OS's normal system call interface and do not emulate hardware. This form of virtualization is not as flexible as other virtualization approaches since it cannot host a different guest OS or kernel as compared to host.

2.1.1 Linux Containers

LXC (Linux Containers) is an OS-level virtualization method for running multiple isolated Linux systems on a single host. OS-level virtualization does not provide a virtual machine, but rather a virtual environment that has its own CPU, memory, file system view, and network.

Kernel features like namespaces, cgroups, and chroot enforce isolation among the containers. A namespace wraps a particular global system resource in an abstraction such that the processes within the namespace get their own isolated instance of the global resource. Linux currently has six different types of namespaces; mount, UTS, IPC, PID, network and user namespace. All these namespaces together form the basis of containers. Performance isolation within containers is ensured via cgroups. Cgroups (control groups) is a Linux kernel feature that limits and isolates the resource usage (CPU, memory, disk I/O, network, etc.) of a collection of processes. For example, restricting the amount of memory or the number of CPUs used by a particular container.

Figure 2.1 consists of a host OS with its own kernel data structures and hardware

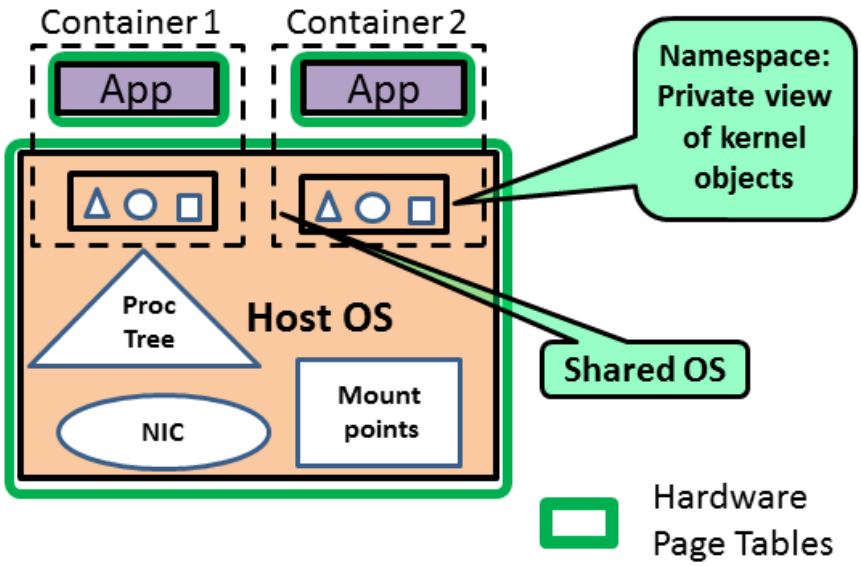

Figure 2.1: [7] OS Level Virtualization (Containers)

page tables. The host OS consists of two instances of containers. Each container shares the same kernel and hardware page tables as that of the host and provides an application its own view of OS. A single application can run in a container without the overhead cost of running a type 2 hypervisor. Multiple isolated containers can also run on a single host as shown in the Figure 2.1. Containers supposedly have near-native performance when compared to VMs because containers do not emulate hardware. These isolated environments run on a single host and share the same kernel and do not allow running different OS or kernel versions.

2.1.2 Solaris Containers

Solaris containers [20] with resource controls are known as *zones*. There are two types of zones: *global* and *non-global* zones. The traditional view of a Solaris OS is a global zone with process ID 0. It is the default zone and is used for system wide configuration. Each non-global zone shares the same kernel as that of the global zone. Non-global zones have their own process ID, file system, network namespace, are unaware of the existence of other zones; can have their own time zone setting and boot environments. Non-global zones are thus analogous to LXC.

2.1.3 FreeBSD Jails

Jails [3] are a chroot extension and provide OS-level virtualization on FreeBSD. Jails have Kernel tunable (sysctl) parameters that can limit the actions of the root user of that jail. Each Jail in FreeBSD has a directory sub-tree, a hostname and an IP address. Jails have their own set of users and root account which are limited to the jail

environment. The root account of a jail is not allowed to perform operations outside of the associated jail environment.

Thus, there are various OS-level virtualization techniques on different OS that are more or less extension to chroot environments.

2.2 Hardware Based Virtualization

Hardware based virtualization provides a way to emulate the hardware so that each VM has an impression that it is running on its own hardware. Hardware vendors are developing new features to simplify virtualization techniques. Some examples are Intel Virtualization Technology (VT-x) and AMDs AMD-V which have new privileged instructions, and a new CPU execution mode that allows the VMM(hypervisor) to run in a new root mode. Hypervisor is a piece of software that facilitates creation and management of VMs.

There are generally two types of hypervisor:

- **Type 1: Native or Bare-Metal Hypervisor**: This type of hypervisor runs directly on the host's hardware. Guest OS can be installed on top of this hypervisor. Such hypervisors have lesser memory footprint as compared to type 2 hypervisor. Examples of bare metal hypervisor are Citrix XenServer, VMware ESX, and Hyper-V.

- **Type 2: Hosted Hypervisor**: This type of hypervisor requires a base OS that acts as a host. Such hypervisors abstract the presence of host from the guest OS. They may use hardware support for virtualization or can just emulate the sensitive instructions using binary translation. Examples of hosted hypervisor are VMware Workstation, VirtualBox, and KVM.

Figure 2.2 shows a hypervisor with two instances of VM running on it. Each virtual machine runs its own copy of a legacy OS, which in-turn has application running on it. The hypervisor provides another set of page tables for example, Extended Page Tables (EPTs), which provide an isolated view of guest-physical memory, and allows an unmodified guest to manage its own address translation. Virtual machines provide better security isolation to the applications within the VM as compared to containers.

KVM (Kernel based Virtual Machine) is a full virtualization solution for Linux on x86 hardware and consists of virtualization extensions (Intel VT or AMD-V). KVM is a type 2 hypervisor which uses Linux as the host OS. KVM consists of a loadable kernel module, kvm.ko, which provides the core virtualization infrastructure, and a processor specific module, kvm-intel.ko or kvm-amd.ko. Each KVM VM on Linux host is a user space QEMU process. KVM arbitrates access to the CPU and memory, and QEMU emulates the hardware resources (hard disk, video, USB, etc.).

Throughout the thesis, we use Linux's KVM [31] and LXC [6] as representative examples of both technologies. We selected KVM and LXC in part because they can use the same kernel, eliminating possible experimental noise. KVM has a comparable design to other type 2 (hosted) hypervisors; further, KVM's content-based deduplication has a first-order effect on memory overhead and is comparable to other VMs such

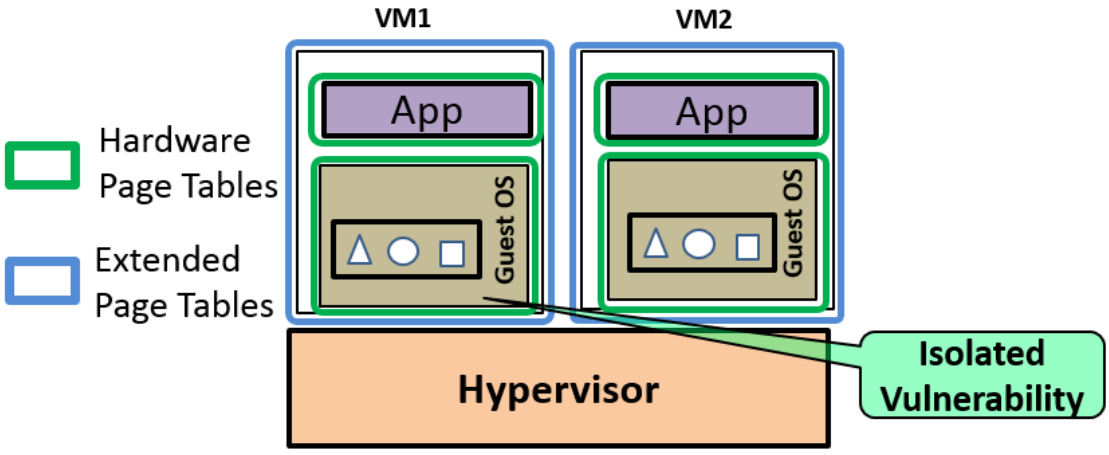

Figure 2.2: [7] Hardware Based Virtualization

as Xen, VMware, and VirtualBox. The design of LXC is also similar to other container implementations.

Security Isolation Problem of Containers: The difference in implementation techniques of containers as compared to VMs, raises concerns about security. Unlike VMs, containers expose the host system call table to each guest, and rely on pointer hooks to redirect system calls to isolated data structure instances, called namespaces in Linux. One security concern for containers is that there may be exploitable vulnerabilities in the pointer indirection code, leading to information leakage or privilege escalation. System calls servicing one guest operate in the same kernel address space as the data structures for other guests. For this reason containers also disallow functionality such as loading kernel extensions. A second security concern for containers is that any vulnerabilities in the system call API of the host kernel are shared, unlike VMs. Specifically, a kernel bug that is exploited through a system call argument is a shared vulnerability with a co-resident container, but not on a co-resident VM. As a point of reference, the national vulnerability database [41] lists 147 such exploits out of 291 total Linux vulnerabilities for the period 2011–2013. In short, containers inherit the same security problems as monolithic operating systems written in unsafe languages, which caused people to turn to hypervisors for security isolation. In contrast, the interface exported by a shared hypervisor is narrower, and less functionality executes in an address space shared among guests.

2.3 Memory Types

A process accesses physical memory for two major purposes, either to read files or store the non-persistent runtime data, based on its use. Depending on these usage patterns

the physical memory can be broadly classified into two types:

- **Anonymous Memory:** The type of memory used by the heap and stack segments of a running process is known as anonymous memory. For example stack, segments created and modified at run time, data segments which are loaded from the binary, heap segments and the shared pages which are mapped via shmem. Anonymous memory contents cannot be evicted from main memory without specifically writing them to a dedicated disk area on a background store (i.e., pagefile, swap partition)
- **File Backed/Named Memory:** The pages that cache the contents of files or disk blocks in the RAM page cache constitute file backed memory.

2.4 Deduplication Methods

This section describes various deduplication techniques available for different OS or virtualization solutions. Deduplication plays a key role while analyzing the memory footprint of VMs and containers and also affects the consolidation density of guests.

KSM (Kernel Samepage Merging) [15] is a memory deduplication approach in Linux that runs a single thread to scan and merge duplicate main memory pages based on their content. KSM is a Linux kernel module that allows sharing duplicate anonymous memory across different processes and KVM virtual machines.

To find duplicate pages in the system it uses two trees, a *stable tree* and an *unstable tree*. The stable tree consists of shared pages generated by KSM, whereas the unstable tree consists of pages which are not shared, but are tracked by KSM to find duplicates.

KSM only regards specifically advised pages (madvise syscall) as mergeable. An application uses madvise to tell the kernel how it wants to use some mapped or shared memory areas so that the kernel can choose appropriate read-ahead and caching techniques. MADV_MERGEABLE flag enables KSM tracking. The kernel regularly scans the areas of user memory that are marked as mergeable for identical content. The reference to such a page is marked as COW (Copy-on-write). If a process updates the content of such a page then it is automatically copied and the changes are applied on the new page. KSM only merges anonymous pages and is intended for applications that generate many instances of the same data (e.g., virtualization systems such as KVM). QEMU is KSM-enabled by default. In this thesis, we use KSM as a technique for deduplicating memory contents across VMs hosted over a type 2 hypervisor, i.e., KVM, which helps in reducing the memory footprint of VMs considerably.

UKSM [9] or Ultra KSM is a modified version of KSM with better scanning and hashing algorithms. It scans anonymous VMAs of applications of the entire system instead of applications that are KSM enabled. Thus, all programs can benefit from UKSM. Xen-dedupe [11], based on core algorithms of UKSM, has further improvements which can efficiently scan the Xen guests' (domU) virtual memory and merge the duplicated pages. However, UKSM is a newer implementation, has no published results and is also not integrated in the mainline kernel.

Page Fusion [10] technique is specifically used with VirtualBox, a type 2 hypervisor. VirtualBox Page Fusion efficiently determines the identical memory pages across VMs and shares them between multiple VMs as a COW page. However, it is supported only on 64 bit hosts with Windows guests (2000 and later). This technique uses logic in the VirtualBox Guest Additions to quickly identify memory cells that are most likely to be identical across VMs. This technique achieves most of the possible savings of page sharing immediately and with very little overhead. Page Fusion can only be controlled when a guest has shut down, as opposed to KSM in Linux which deduplicates memory for running guests as well. Another difference between KSM and VirtualBox Page Fusion is that Page Fusion does not consider the host memory while scanning for duplicates and only considers common memory areas across VMs. Therefore, possible memory savings of an identical host and guest kernel are missed by Page Fusion.

Windows Server 2012 R2 supports deduplication of actively used VHDs. The deduplication support is only for VMs used as a part of a Virtual Desktop Infrastructure (VDI) deployment. Deduplication for VHD files used by VMs running a server OS is not supported thus narrowing the scope for deduplication among guests.

Content Based Page Sharing [49] is used by VMware ESX server, which is a type 1 hypervisor. The redundant page copies are identified by their content and can be shared regardless of when, where, and how those contents are generated. ESX hypervisor scans the content of a guests' physical memory for sharing opportunities. It uses hashing techniques to identify potential identical pages instead of comparing each byte of a candidate guest physical page to other pages. This technique considers all memory pages irrespective of anonymous or file backed. However, there are fewer file-backed pages (from the host perspective) in a type 1 hypervisor as compared to a type 2 hypervisor. KSM, is quite analogous to this technique except for the fact that it scans only anonymous memory regions and currently works with a type 2 hypervisor, KVM.

Thus, this Chapter enhances the understanding of thesis by providing background details on virtualization technologies, memory types, and deduplication techniques. Nearly all hypervisors have some sort of deduplication, but the details vary in terms of the scope and level of hints required. However, we believe that KSM is a reasonably representative deduplication technology.

Chapter 3

Related Work

Our study compares VM and container overheads with respect to consolidation density and latency, and studies the memory footprint of both the virtualization techniques. This thesis suggests ways to reduce the memory footprint of VMs, to make them as lightweight as containers. Thus, the first section of the chapter outlines the studies which compare VMs and containers, and the second section outlines the studies that explore memory sharing opportunities with respect to the VMs.

3.1 VMs Vs. Containers

A few previous studies have evaluated the benefits of containers relative to VMs in specific contexts; this paper bridges gaps in previous evaluations of both approaches.

Felter et al. [23] measure CPU, memory throughput, storage and networking performance of Docker and KVM. Commensurate with our unoptimized measurements, this study reports KVM's start-up time to be 50× slower than Docker. This study concludes that containers result in equal or better performance than VM in almost all cases and recommend containers for IaaS. While our results corroborate the findings of this study with different benchmarks, this study does not measure memory footprint or scalability of either system. This study overlooks the *checkpoint and restore* optimization to start a VM, which brings KVM start-up time from 50× to 6×.

Another study measures the memory footprint of Docker and KVM using OpenStack [5] and figures out that while running real application workloads, KVM has 6× larger memory footprint than containers. This study does not analyze to what degree this 6× is fundamental; our results indicate that this could likely be reduced.

Regola and Ducom [44] have done a similar study of the applicability of containers for HPC environments, such as OpenMP and MPI. They conclude that I/O performance of VMs is the limiting factor for the adoption of virtualization technology and that only containers can offer near-native CPU and I/O performance. This result predates significant work to reduce virtual I/O overheads [13, 14, 26, 34, 48].

Xavier et al. [50] compare the performance isolation of Xen to container implementations including Linux VServer, OpenVZ and Linux Containers (LXC). This study concluded that, except for CPU isolation, Xen's performance isolation is considerably

better than any of the container implementations. Another older performance isolation study [36] reaches similar conclusions when comparing VMware, Xen, Solaris containers and OpenVZ. We note that these results may be dated, as the studies used Linux versions 2.6.18 (2006) or 2.6.32 (2009).

Soltesz et al. [46] measure relative scalability of Linux-VServer (another container technology) and Xen. They show that the Linux-VServer performs 2× better than VMs for server-type workloads and scale further while preserving performance. However, this study only measures aggregate throughput, and not the impact on a single guest—a common scenario in a multi-tenant environment for achieving Quality of Service for each tenant. To measure scalability, we also study the effects of packing the host with more guests. However, we consider the performance of just one guest in such a scaled environment instead of an aggregate throughput of all the guests.

Concurrent with this work, Canonical conducted some similar experiments comparing LXD and KVM running a complete Ubuntu 14.04 guest (a slightly older version than this paper) [21]. Their density measurements show container density 14.5× higher than KVM, limited by memory, whereas our measurements show the cost relative to LXC at 11×. It is unclear which differences in the experiments account for the overall differences in density. The start-up times reported for both LXD and KVM are considerably higher than our measurements, and these measurements do not include the checkpointing optimization. The article reports a 57% reduction in network message latency for LXD, which we did not measure.

Clear containers [1] is a concurrent work which aims to reduce the overheads of VMs as compared to containers. This work optimizes away a few early-boot CPU initialization delays, which reduces start-up time of a VM. Clear containers is a system where one can use the isolation of virtual-machine technology along with the deployment benefits of containers. It is an ongoing work and some of our techniques along with the improvements made by this work can help reduce the overheads of VM to a large extent.

3.2 Memory Sharing Opportunities

Our work focuses on analyzing the memory sharing opportunities, both for VMs and containers, as well as identifies ways of reducing the memory footprint of VMs. A few research works have looked into the problem of reducing the memory footprint of a VM.

Memory deduplication increases the memory density across actively-used memory in the system. Deduplication also helps in pushing back the boundary at which the system starts thrashing. There is much sharing potential between multiple VMs running on a host, which often end up having equal pages within and across the VMs. Such redundant memory can be collapsed by sharing it in a COW fashion. For host, the guest's entire memory is anonymous. *Paravirtualization* thus helps to close the semantic gap between the host and the guest by providing an interface of communication between host and guest. However, it comes at a cost of modifying both host as well as the guest.

Transparent page sharing was the first memory saving mechanism pioneered by Disco [19]. It is a mechanism for eliminating redundant copies of pages, such as code or

read-only data, across VMs. Once copies are identified, multiple guest "physical" pages are mapped to the same machine page, and marked copy-on-write. Writing to a shared page causes a fault that generates a private copy of the page. However, Disco required several guest OS modifications to identify redundant copies as they were created.

Thus, page sharing in hypervisors can be broadly classified in two categories:

- **Content Based Scanning Approach:** It periodically scans the memory areas of all the VMs and performs comparisons to detect identical pages. Inter VM content based page sharing using scanning was first implemented in VMware ESX server [49]. It scans the content of guest physical memory for sharing opportunities and uses hashing to identify potential identical pages. If a match is found, a byte-by-byte comparison is done to eliminate the false positives.

  Diwakar et al. [27] provide an extension for Xen VMM, Difference Engine, which leverages a combination of whole page sharing, page patching and compression to realize the benefits of memory sharing. This model leads to 1.5% and 1.6-2.5% more memory savings than VMware ESX Server for homogeneous and heterogeneous workloads respectively. Difference engine can further compact shared memory, and work around smaller differences, whereas our analysis tries to understand how unique pages are being used.

- **Paravirtualized Approach:** It is an alternative to scanning based approaches where duplicate detection happens when the pages are being read in from the virtual disks. Here the virtual/emulated disk abstraction is used to implement page sharing at the device level itself. All VM read requests are intercepted and pages having same content are shared among VMs.

  Grzegorz et al. [40] successfully merge named pages in guests employing sharing aware virtual block devices in Xen which detect sharing opportunities in the page cache immediately as data is read into memory. Guests are modified to use the additional memory and give it off when required. However, this approach requires modifying the guests. Our work identifies solutions to lower the memory footprint by not modifying the guests and also explores sharing opportunities beyond anonymous memory regions.

3.3 Memory Deduplication Scanners

Such scanners mitigate the semantic gap by scanning the guest pages for duplicate content, indexing the contents of the memory pages at a certain rate irrespective of the page's usage semantics. For example, KSM, which is a content based memory deduplication scanner in Linux. However as opposed to KSM, which is an I/O agnostic scanner, Konrad et al. [38, 39] extend the main memory deduplication scanners through Cross Layer I/O-based hints (XLH) to find and exploit sharing opportunities earlier without raising the deduplication overheads. It merges equal pages, which stem from the virtual disk image earlier by minutes thereby saving 4 times as much memory as KSM. This approach does not modify the guest and uses a bounded circular stack approximately

2 MB to hold the hints. A non I/O process never starves and if no hints are there then the linear scan takes place and is same as a default KSM implementation. For a kernel build workload with 2 VMs 512 MB each they are able to save 250 MB memory as compared to 75 MB. Thus, XLH looks into ways of deduplicating memory by intercepting I/Os from virtual disks and comparing with already existing pages from other disks. Our work identifies other avenues like deduplicating file-backed vs anonymous memory, file-backed vs file-backed memory which along with XLH can ensure higher memory savings.

Jian et al. [29] describe a framework wherein the guests are modified to share memory with the host by giving the unused guest memory directly to the free memory list in the host MMU without any changes to the host's KSM or swap algorithm. This helps in eliminating a number of possible page swapping and merging in VMs and reclaiming unused memory from the virtual machines. It also helps in enhancing the compute density and reduces the energy cost of the whole system.

Sharma and Kulkarni [45] implement an exclusive host page cache for KVM which proactively evicts the redundant pages from host page cache and maintains a single copy of such pages across the host and the guests. It helps in solving the problem of double caching, thereby saving a higher amount of memory (savings 2-4x) so as to provision more VMs on a host with the same RAM. It also optimizes the existing KSM duplicate page detection by replacing search trees with hash tables and full page comparisons with checksum (jhash2) comparisons. This work improves KSM performance while we look beyond the sharing opportunities presented by KSM, and explore other potential venues for reducing memory footprint of VMs.

Kloster et al. [32] present an approach to retrieve information on how redundant pages are used in virtualized systems. They conclude that greater the degree of homogeneity in the virtualized system, greater is the level of redundancy. The main factor causing redundancies are application binaries and common kernels. They use a driver for the Xen modified Linux kernel running on the Xen VMM patched with CBPS and Potemkin. A new contribution above this work is a cost and benefit analysis of containers and VMs. This thesis also identifies the ways to reduce the incremental cost of VMs by analyzing the memory sharing potential left unexplored by KSM. Instead of using Xen, we use KVM VM and unmodified guests in our study.

With respect to reducing the start-up time of VMs, SnowFlock [33], a VM Fork system reduces the boot-up time of a newly forked VM to 600-700 ms. On a fork, instead of copying the entire state, a VM descriptor containing VM metadata and guest kernel memory management data is copied to the newly created child. The remaining data is copied on demand. Such techniques can be very useful for quickly spinning up VMs on demand. This is also a profitable option for faster start-up time.

Barker et al. [16] study the dynamics of page sharing to maximize its benefits. Their study shows that the majority of the sharing potential is attributable to the redundancy within a single machine rather than different machines. However in our analysis we focus on inter VM sharing opportunities, which are commensurate with their findings wherein the maximum sharing potential (80%) for two 64 bit Ubuntu base systems is due to inter VM sharing rather than self sharing. They also highlight the ASLR has a potential to affect the level of page sharing. Two VMs that may be

running identical software may have different memory contents if ASLR is enabled, resulting in less possible sharing. They show that the overall sharing was reduced by 10% using the stock Linux kernel. However, from a security point of view, and for a realistic measurement we do not turn off ASLR in our experiments.

Chen et al. [22] propose a lightweight page *Classification Memory Deduplication (CMD)* approach to reduce futile comparisons while detecting page sharing opportunities, as happens in KSM. Since KSM only maintains two global comparison trees, stable and unstable trees, for all memory pages, it results in comparison of a candidate page with many uncorrelated pages in the global trees. In CMD model pages are grouped by access characteristics like write access count and write access distribution of sub pages. The large global comparison trees are divided into multiple trees with dedicated local ones in each classification. Page classifications are performed in the same classification and pages from different classifications are never compared. As compared to baseline KSM, CMD detects page sharing opportunities by more than 98%, with 68.5% fewer page comparisons. However, our work explores memory deduplication potential available beyond what KSM could explore and also suggests some other ways of bringing down a KVM VM memory footprint, e.g., by removing unused devices.

Thus, there are research works which try to maximize the efficiency of KSM by reducing its overheads, or introducing hints so as to advise more pages as mergeable. This work identifies additional opportunities which could possibly reduce the memory footprint of a KVM VM.

Chapter 4

Density

Density refers to the number of guests per physical host. Higher density without compromising the performance of other guests ensures lower cloud computing costs. A data centre operator will benefit more if the guest technology used while setting up the data centre infrastructure is scalable and does not affect the performance of other guests. Therefore, it is important to analyze carefully how the increase in number of guests affects the CPU and I/O performance of other guests on the same physical infrastructure.

Increasing the number of guests can cause the performance of each guest to deteriorate. Therefore, this work quantifies the deterioration of performance for VMs and containers on running more guests than available cores in the host. Running more guests than the host cores can cause a lot of resource contention and swapping between the guests. Thus, if the overheads are high, density of guests should be kept low.

To analyze the impact of density on CPU and I/O performance, we measure the change in performance of one guest running a CPU-bound and I/O bound workload as the number of guests per core increases. The guests that are added incrementally run a busy work and are not idle. SpecCPU 2006 [28] is used for CPU benchmarking and Filebench [24] is used for I/O performance benchmarking.

4.1 Experiments and Analysis

Container Configuration: A container can be configured in two modes: one that runs all of the typical daemons of a fully-functional system (called "full"), and another that runs a single application in a sandbox (called "single"). Because the "full" configuration is comparable to one or more VMs running the same guest OS, this paper primarily studies this comparison, although some data is also reported for the "single" case, as the "single" configuration is also a popular option. In general, the "single" case is more fairly compared to a library OS [35, 43, 47] or a sandbox system [25, 51].

Experimental Environment: All experiments are conducted on a Dell Optiplex 790 with a 4-core 3.40 GHz Intel Core i7 CPU, 4 GB RAM, and a 250 GB, 7200 RPM ATA disk. The host system runs Ubuntu 14.10 server with host Linux kernel version

3.16. Host system includes KSM and KVM on QEMU version 2.1.0 and LXC version 1.1.0. Each KVM Guest is configured with 1 virtual CPU with EPT confined to a single host CPU core, 1GB RAM, a 20GB virtual disk image, Virtio enabled for network and disk, bridged connection with TAP, and runs the same Ubuntu and Linux kernel image. Each LXC guest is configured with 1 CPU core confined to a specific host CPU core. For a fair comparison, we limit cgroups memory usage of LXC guest to 256 MB, which is equivalent to the resident size of a typical VM (248 MB).

Unless specified, KSM is enabled and the *ksmd* daemon scans memory areas to search for duplicates. KSM configuration has been modified from its default for all the experiments conducted as part of this work. To make the KSM numbers stabilize soon and explore the existing sharing potential to its best, the number of pages to be scanned is set as 500 and sleep time is set as 10 ms by modifying the respective parameters corresponding to KSM in *sysfs*.

4.2 CPU-Bound Workloads

We measure the performance of two SPEC CPU2006 benchmarks configured with the standard base metric configuration that enforces strict guidelines for benchmark compilation. One benchmark simulates a quantum computer (462.libquantum) and one simulates a game of chess (438.sjeng). We chose the quantum computing and chess benchmarks from the SpecCPU suite because they perform CPU intensive integer operations and minimal I/O operations. We run the benchmark on the host to get the base numbers. Next, we run it on one guest, while all other guests increment an integer in a while loop, simulating an environment where all guests are running CPU-bound workloads. We run one iteration of the benchmark to warm up the system, and report mean and 95% confidence intervals for subsequent runs. We compare both containers and VMs with densities of 1, 2, 4, and 6 guests/core (i.e., 1, 4, 8, 16, and 24 guests on a 4 core system).

SPEC CPU2006 462.libquantum benchmark gives results with reference run time, base run time, and the base ratio values. SPEC uses a reference machine to normalize the performance metrics used in the CPU2006 suites. Each benchmark is run and measured on this machine to establish a reference time for that benchmark. This reference time is compared against the base run time (time taken to run the workload on our experimental setup) to get the base ratio. Higher reference to the run time ratio ensures better CPU performance. For sake of simplicity, we just use the run time in seconds to understand the impact of increasing density on performance of the guest executing the CPU-bound workload.

Figures 4.1 and 4.2 show the execution time of the SPEC CPU2006 libquantum and sjeng benchmarks as guest density increases, where lower is better. With only one guest on the system, performance of both a VM and container are comparable to a native process. As density increases, both–VMs and containers–performance degrades equally. In general, the differences in execution time are very small—8.6% in the worst case—and often negligible. In summary, for CPU-bound workloads at reasonable densities, VMs and containers scale equally well.

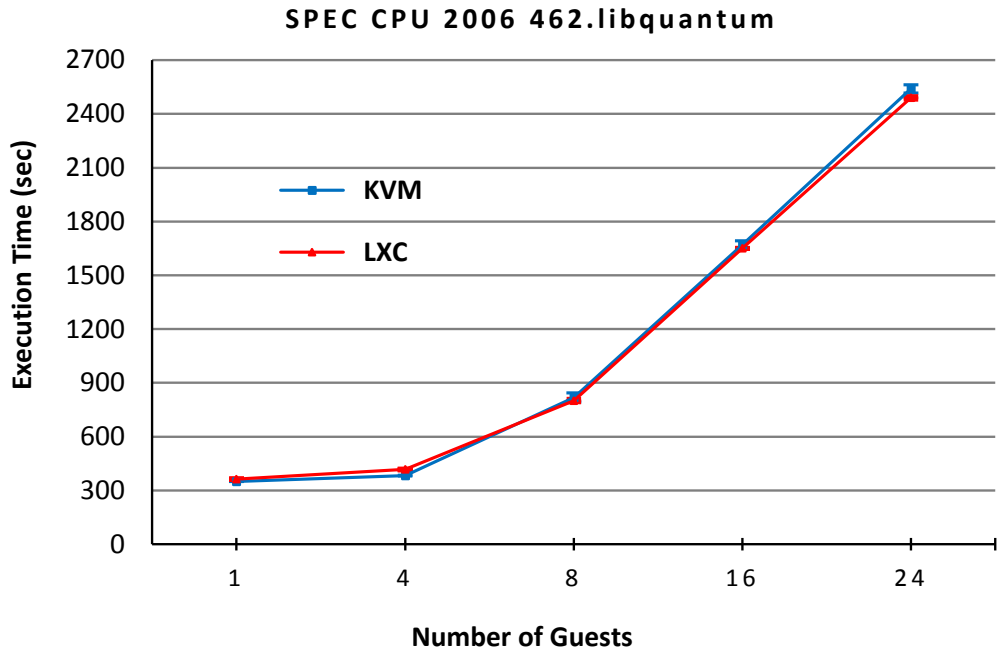

Figure 4.1: SPEC CPU2006 execution time for simulating a quantum computer as number of guests increases on a 4-core machine. Lower is better.

For a CPU-bound workload, it doesn't make much difference, although containers are slightly better at high degrees of over-commitment ($\geq$ 3 vCPUs per core). This shows that the hardware virtualization features can run a VM at almost native speed like containers as long as there is no I/O to cause VM exits and the scale-out performance of VMs and containers is nearly equivalent to each other.

4.3 I/O-Bound Workloads

We use Filebench [24], a file system and storage benchmark, to measure the performance of a typical fileserver workload running in one guest. The fileserver workload measures the performance of various file operations like stat, open, read, close, delete, append and write in terms of operations per second (ops/sec). We plot the mean ops/sec and 95% confidence intervals of containers and VMs in case of 1, 2, 4, 6, and 8 guests/core. The other guests are running the same busy workload which increments an integer. Here, we measure the effect of CPU-bound workload running on other guests, over the I/O workload running via Filebench, as the number of guests increases.

Figure 4.3 shows that the deterioration in performance is proportionate for the KVM and LXC. However, KVM performance is lower than LXC by a near-constant factor. This difference is attributable to the cost of a VM exit, which is incurred every few I/Os when using virtio.

We also evaluated the effect of running an I/O bound background workload on

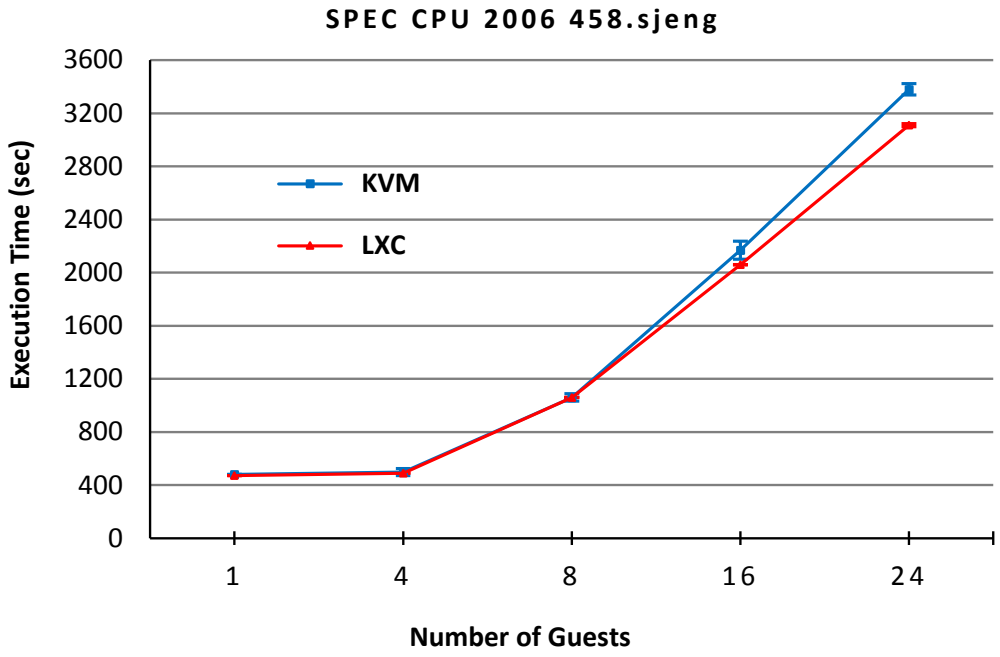

Figure 4.2: SPEC CPU2006 execution time for chess simulation as number of guests increases on a 4-core machine. Lower is better.

other VMs. Unfortunately, the system could not finish the benchmark with more than 2 VM guests/core. We suspect this is a bug in virtio's memory management, not a performance cliff, as the system appeared to be thrashing on memory. The workload did complete on containers. For up to two guests per core, the performance trend for containers and VMs with concurrent I/O was commensurate to the trend with CPU-bound background work.

We hasten to note that I/O pass-through, where a device is directly mapped into a VM, can be used to eliminate these VM exits [13, 14, 26, 34]. Recent works report near-equivalent I/O performance to bare metal, using these techniques [26, 48]. ELI [26], directly maps the devices in the guest by using a shadow IDT for the guest. ELI, shows improvement in performance of I/O intensive workloads–Netperf 98% and Apache 97% of the bare metal throughput. ELI, work is also based on KVM, however it uses older Linux kernel and is not implemented in the latest KVM code base. An ELI patch to the latest KVM code base can improve the I/O performance of KVM VMs. These works require some cooperation for interrupt management between the guest and host that could be hard to secure on current hardware; it is likely that minor changes to virtualization hardware could obviate any security issues and minimize VM exits for I/O interrupts. Tu et al. [48] present DID, a comprehensive solution to the interrupt delivery problem on virtualized servers. DID completely eliminates most of the VM exits due to interrupt dispatches and EOI notification for SRIOV devices, para-virtualized devices, and timers. The current DID prototype is built into the KVM hypervisor that supports direct pass-through for SRIOV devices. The DID prototype reduces the number of VM

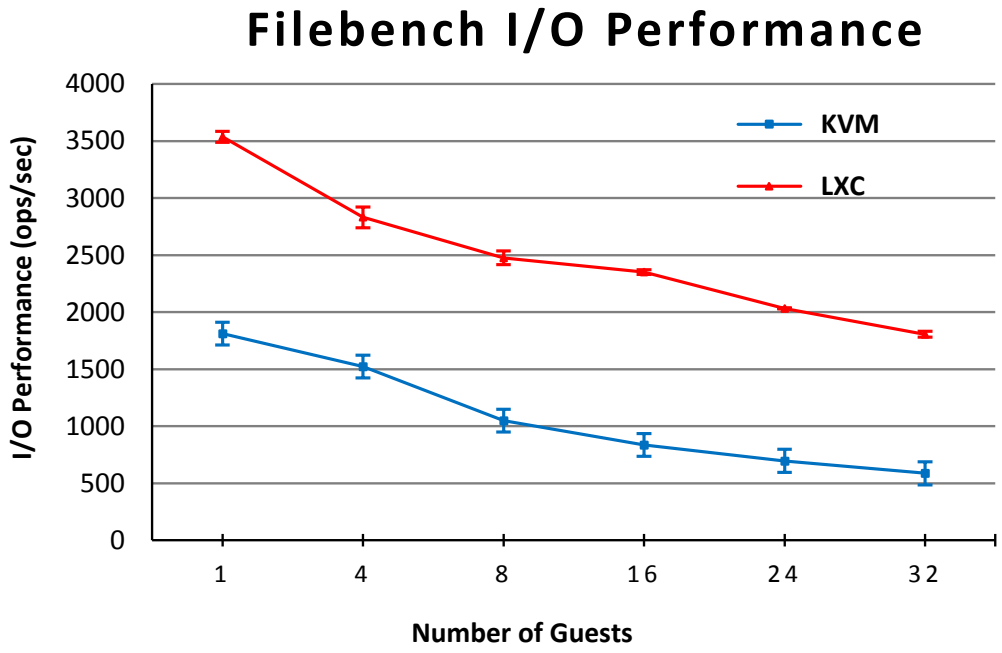

Figure 4.3: Filebench I/O performance as number of guests increases on a 4-core machine. Higher is better.

exits by a factor of 100 for I/O-intensive workloads. Moreover, recent studies indicate that the best performance is likely achieved when I/O drivers are pushed into the application [17, 42], which would ultimately obviate any difference between containers or VMs with respect to I/O.

Thus, our results, as well as the results of other researchers, indicate that performance of I/O-bound workloads is determined primarily by how devices are multiplexed among applications and guests. Direct-to-application I/O will likely yield the highest performance, and we expect that both containers and VMs can implement this model.

Chapter 5

Start-Up Latency

Start-up latency is the time required to provision a guest to service a request. It affects the efficiency of cloud computing. Lower latency of a guest helps a data centre operator to provision guests for the peak load and thereby ensures higher chances of meeting response deadlines as specified in service level objectives (SLOs). Therefore, in this chapter, we study the start up time for two guest technologies–VMs and containers– and evaluate if any optimization can further help to reduce the start-up time of the guests.

Intuitively, containers should have a lower start-up latency than a guest OS, as only API-relevant data structures need to be initialized, eliding time-consuming boot tasks such as device initialization. Start-up time for a VM is the time required to boot a VM till a login screen appears, and for a Linux container, start-up time is the time required to run the lxc-start command. We account for the start-up times to get an estimate of the start-up latency while these guest technologies are used in data centre deployments. We measure the start-up time of VMs and containers as stated in Table 5.1. While it takes 10.3 seconds to boot a VM, it takes 0.2 seconds to start a container. Thus, a typical VM takes 2 orders of magnitude longer to start up than a container.

The delay in the start-up time of a VM is due to booting an entire OS, which is not the case with containers. Containers share the same kernel as that of the host, so they save upon the time spent by the bootloader and other initialization tasks. If we can start VMs in real-time without having actually to boot the OS, the VM's start-up time can be reduced significantly. The cloud providers can create surplus VMs offline and state-save these surplus VMs so that new VMs can be started as required in a short period.

Start up time of a VM or container is significant for applications that scale out to handle peak workload by spawning more instances of the application. In such a scenario, it is faster to start more containers and start servicing the load than to start more VMs. As a result, the cloud providers prefer containers to VMs to handle peak load.

Running a single-application container cuts the start-up time of LXC in half, but indicates a considerable start-up cost that is independent of the number of processes. We also measure the start-up time of a typical FreeBSD Jail, which starts fewer processes than a full Linux instance and thus takes roughly half the time to start.

Guest	Start-Up time(sec)	Restore time(sec)
KVM	10.342	1.192
LXC	0.200	0.099
LXC-bash	0.099	0.072
FreeBSD Jail	0.090	-

Table 5.1: Start-Up and Restore time for different guests

However, there is at least one other alternative when one is running the same kernel on the same hardware: *checkpoint a booted VM* and simply *restore the checkpoint after boot*. In case of a VM, this elides the majority of these overheads, reducing the start-up time by an order of magnitude. The same optimization can also be applied to containers by use of CRIU. CRIU [2] Checkpoint/Restore In User space is a software tool for Linux OS, which freezes a running application (or part of it) and checkpoints it to a hard drive as a collection of files. One can then use the files to restore and run the application from the check pointed state. In contrast, for a container, the savings of checkpoint/restore are more marginal (0.008s). Starting a container is almost as fast as restoring it from a saved state. Thus, checkpoint restore optimization does not give the same benefits to containers as that to a VM.

Even with the checkpoint/restore optimization, there is still a factor of 6× difference in the start-up latency of a VM and container. However, there are likely opportunities to further close this gap, and we expect that optimizations to reduce the restore time will disproportionately help VMs. For instance, the SnowFlock [33] VM Fork system reduces the start-up time of a subsequent Xen VM by 3×—4× difference by duplicating just the VM descriptor and guest kernel memory from an already running VM and then duplicating remaining data on demand. Bila et al. [18] analyze the mean working set of a 4 GB VM to be just 165 MB and as a result, SnowFlock can reduce the start-up by just duplicating the working set size of a running VM.

Although it is hard to do a precise analysis from outside a data center and on different hardware, we observe that adding a 100-200ms delay to a request is unlikely to be satisfactory simply given the orders of magnitude for request processing in recent data center performance analysis studies [30, 37]. For instance, in the BigHouse workload model, this would increase service times from at least a factor of 2, up to 2 orders of magnitude. It is unlikely that any technology with start-up times measured in hundreds of milliseconds will ever facilitate demand loading of latency-sensitive services. We suspect that more radical changes to the OS, such as a library OS or more aggressive paravirtualization, will be required to realistically meet these requirements. As a point of comparison, the Graphene library OS paper reports that a Linux process can start in 208 microseconds and Graphene itself can start in 641 microseconds [47].

Thus, when checkpoint/restore time for VMs is considered, containers do a have a clear advantage in start-up time, but the gap is smaller than one might expect. In both cases, the costs may be too high for anything other than provisioning for the peak demand. Broadly speaking, the start-up time for a VM, which is 50× that of a container can be reduced to a factor of 6× by considering *save state and restore optimization*. A similar optimization does not benefit containers and thus the gap between the start-up latency of VMs and containers is not as high as is purported.

Chapter 6

Memory Analysis

Memory footprint determines consolidation density. If a guest technology has high memory footprint, then the consolidation density will be low on a physical host with fixed RAM. Density is a critical factor for data centre operator and determines the efficiency of cloud computing. Therefore, a data centre operator would benefit from a guest technology that has low memory footprint, ensuring more guests can be run on a physical host with fixed RAM. This Chapter provides a background on memory footprint and memory measurement metrics; describes memory analysis of VMs and containers; studies the incremental cost of a VM and suggests three possible solutions to reduce the incremental cost of a VM which can help reduce the memory footprint gap between VMs and containers.

6.1 Memory Footprint

Memory footprint of a process is the amount of physical memory occupied by that process. For a VM, the memory footprint is the amount of physical memory occupied by the qemu-kvm process which includes memory occupied by its stack and heap, shared libraries which are required to run a QEMU process, devices emulated by QEMU for the guest, and the guest RAM, as shown in Figure 6.1. For a container, memory footprint is the amount of physical memory occupied by all the processes running within the container.

Several metrics to measure the memory footprint of a process are as follows:

- **RSS (Resident Set Size):** It is the most commonly used measure of per-process memory under Linux, reported by utilities such as *ps* and *top*. It indicates the sum of the size of all the pages mapped by the specified process including all shared pages. Thus, the sum of RSS of all system processes can exceed system RAM.

- **PSS (Proportional Set Size):** It is the sum of the size of all pages mapped uniquely in the process, plus a fraction of size of each shared page. It is a fairer metric as compared to RSS as it accounts for a proportion of a shared page instead of counting it as a whole.

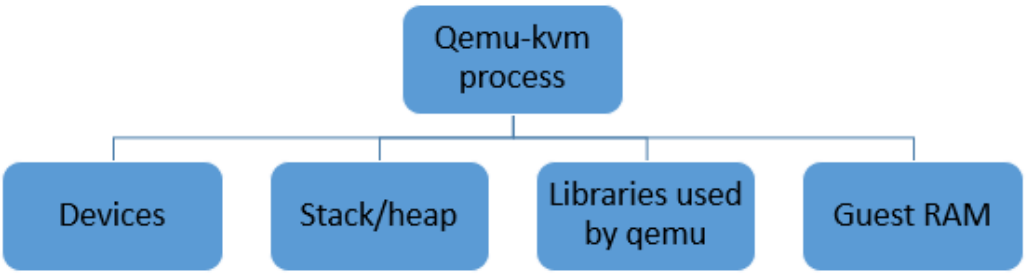

Figure 6.1: Areas of memory utilization within qemu-kvm process

- **USS (Unique Set Size):** It is the sum of the size of all physical pages mapped only in the given process. It is the memory which is given back to the system, when a process terminates.

Table 6.1 shows the various memory metric values for a process $P1$, which has 1000 unique pages mapped to it and 100 pages, which are shared by a process $P2$.

Metric	Calculation	Value in MB
RSS	$(1000 + 100) * 4KB$	4.29 MB
PSS	$(1000 + (100/2)) * 4KB$	4.1 MB
USS	$1000 * 4KB$	3.90 MB

Table 6.1: Memory measurement metrics for Process $P1$

Thus, PSS gives most accurate measurement of the actual memory footprint of a process as the size of a shared page is accounted as a fraction of the processes sharing it and not a complete unit as in RSS.

6.2 Basic Framework

This section describes a basic framework that helps us to analyze the deduplication potential between host and guest, within guests and between two guests. In addition to the basic setup as described in Chapter 2, huge page support is disabled and standard page size of 4KB is used.

6.2.1 PFN and Flags of Mapped Pages in a Process

To get the physical frame number (PFN) of the pages mapped in a process, we read all the VMA regions from the */proc/[pid]/smaps* file as shown in the Algorithm 1.

Each VMA region has start virtual address, end virtual address, device number, an inode number and a corresponding mapping. *smaps* file for a process also contains information about RSS, PSS, MMUPageSize for that VMA. As shown in Algorithm 1, iterate over all the *page_size* pages in the given VMA address range, seek to the *addr* offset in /proc/[pid]/pagemap to get the PFN corresponding to the virtual address and seek to the pfn offset in /proc/kpageflags to retrieve the page flags for that PFN. For a qemu-kvm VM, we obtain the host PFNs and flags for all pages mapped in the QEMU process.

The above information can be used to determine the following:

- Number of physical pages that are mapped in the big VMA region (which points to the guest) of a qemu-kvm process
- Number of physical pages that are mapped in the guest and have the potential to be deduplicated with the rest of the host.
- Number of physical pages in the big VMA region that are mapped and present in EPTs.

Algorithm 1 PFN and page flags of mapped pages in a process

```
procedure PHYSICALPAGEINFO
    fd = open("/proc/[pid]/smaps")
    for each VMA_region ∈ fd do
        start_addr = start virtual address of VMA_region
        end_addr = end virtual address of VMA_region
        page_size = mmu page size of VMA_region

        for each addr ∈ (start_addr, end_addr) do
            pfn = seek to the addr offset in /proc/[pid]/pagemap
            page_flags = seek to the pfn offset in /proc/kpageflags
            addr = addr + page_size
        end for
    end for
end procedure
```

6.2.2 Host Physical Addresses and Hashes

We use Rekall [8], a memory forensics tool to gather the EPT mappings of a KVM guest. This tool has been forked from the Volatility code base and provides a complete memory acquisition and analysis solution for Windows, OS X and Linux. To analyze the deduplication potential of a guest with the host, we need host physical addresses and the hashes of the contents pointed to, by them. As access to /dev/mem was restricted for stricter kernel protection therefore we use Rekall's pmem kernel module which provides a complete access to the system memory. We take the BIOS map at

the boot-up and consider the usable memory regions. We then read in page size (4KB) bytes starting at the physical address offset from the /dev/pmem device created by the pmem kernel module and use sha256 to hash the page contents. Thus, as shown in Algorithm 2, we can get PFNs and their corresponding hashes for the entire system memory.

This algorithm can also help to retrieve the hashes of physical pages mapped in the guest memory corresponding to the physical addresses retrieved from Algorithm 1.

Algorithm 2 Address and hashes for all physical pages present in the host

```
procedure HOSTPAGEINFOANDHASH
    insert pmem.ko
    let pmem = /dev/pmem
    read the BIOS map and get the usable address mappings
    for each usable_addr_range ∈ BIOS map do
        start_addr = start physical address
        end_addr = end physical address
        page_size = 4KB

        for each phys_addr ∈ (start_addr, end_addr) do
            page = read page_size from pmem
            hash = sha256sum page
            phys_addr = phys_addr + page_size
        end for
    end for
end procedure
```

6.2.3 EPT Mappings for a qemu-kvm Guest

We use Rekall, as mentioned above, to retrieve the EPT mappings for a guest. It requires a raw image and a host profile to run. Once Rekall session is launched, *vmscan* command scans for the available guest sessions running and then lists the EPT pointer values corresponding to them. One can switch to any of the guest sessions using *sswitch*[*session_no*] command, and can load the guest profile via *smod* command. To get EPT mappings, iterate over address ranges of the physical address space of the virtualization process which provides us the host physical address to guest physical address mappings as shown in Algorithm 3.

6.3 Cost Analysis

To understand the memory usage of the virtualization technologies– VMs and containers– we first analyze the cost or memory used by VMs and containers using both metrics PSS and RSS to decide which metric gives more accurate results.

Algorithm 3 EPT Mappings for a qemu-kvm guest

```
procedure EPTMappings
    create a Rekall profile for guest and host
    use raw image for e.g. /proc/kcore and host profile to launch Rekall in host
    vmscan in the current host session - gives EPT values for the guest sessions
    sswitch to any of the sessions which are result of vmscan
    if in guest session, load the guest profile using smod
    pslist can show all the guest processes running, if required
    for each guest_phys_addr, host_phys_addr, page_size in
session.physical_address_space.get_available_addresses() do
        write guest_phys_addr, host_phys_addr, page_size to a file
    end for
end procedure
```

PSS of a running VM is the sum of PSS values of all the VMAs belonging to the qemu-system-x86 process, which is a user land process that runs a KVM machine. These values can be obtained from the /proc/[pid]/smaps file corresponding to a QEMU process *pid*. PSS of a container is the sum of PSS values of all the processes running within the container. The PSS value of a process is computed in the same way as that described above for a VM. RSS of VMs and containers can also be obtained using the above methodology by considering the *Rss* parameter in the /proc/[pid]/smaps file instead of *Pss*.

We measure PSS and RSS for both–VMs and containers–and see a striking difference in memory footprint values.

6.3.1 PSS Vs. RSS

PSS of a single idle VM is 235 MB whereas that of a container is 16 MB. This implies that 15 times more containers can be run on the same host as compared to VMs. However, if we use RSS for measuring the memory footprint, RSS of an idle VM is 241 MB whereas that of a container is 44 MB. Considering RSS, it implies that 5 times more containers than VMs can run on the same physical host, which is quite in favor of VMs but is misleading. PSS of containers is considerably reduced as Linux containers share libraries and other files among processes through the host page cache and thus the shared pages get accounted as a fraction not as a whole, thereby reducing the cost of containers. Also, PSS of containers is the sum of the PSS of processes running within the container. PSS of processes that run when a Linux container is started are as shown in Table 6.2.

Thus, PSS gives more accurate memory footprint measure as compared to RSS. Therefore, throughout the study, we use PSS to account for the memory used by VMs and containers.

Process	PSS
lxc-start	1.07 MB
init	1.62 MB
cron	0.59 MB
dhclient	2.93 MB
getty (5 processes)	1.6 MB
rsyslogd and child processes	5.53 MB
systemd-udevd	1.22 MB
upstart-file-br	0.52 MB
upstart-socket	0.52 MB
upstart-udev-br	0.52 MB
Total	**16.12 MB**

Table 6.2: PSS of processes running within Linux Container (LXC)

6.3.2 Effect of KSM on Memory Footprint of Guests

KSM, a hash based deduplication technique which deduplicates common memory areas between processes, marks them COW; affects the memory footprint of a VM. The memory footprint of a VM is reduced to 187 MB as shown in Table 6.3. Almost 48 MB of physical memory gets deduplicated with KSM, which eventually helps in provisioning more guests. Therefore, use of deduplication techniques like KSM while deploying guests can help improve consolidation density and provision more guests on a host with fixed amount of memory.

	Without Optimization	With KSM
PSS of an idle VM	235 MB	187 MB
PSS of an idle container	16 MB	16 MB

Table 6.3: Effect of KSM on memory footprint of VMs and Containers

Containers do not benefit from the same optimization as VMs. The PSS of containers remains same that is nearly 16 MB with or without KSM.

We note that, from the host's perspective, any KVM guest RAM, including the guest's page cache is treated as anonymous memory, allowing deduplication between cached pages of disk images. However, Linux containers do not use KSM to deduplicate anonymous (non-file backed) memory because they are not enabled to make madvise system call as explained in the Chapter 2. However, to determine whether KSM would afford additional benefits to containers beyond file-level sharing, we use KSM preload [4] which enables legacy applications to leverage Linux memory deduplication. We manually analyzed the anonymous pages between the two container instances to find duplicates, by dumping the contents of these pages and hashing them. However, we just found 1 anonymous duplicate page among the processes of two containers. Thus, even if KSM were enabled for containers, the reduction in PSS would likely be negligible.

6.4 Density Analysis of VMs

KSM helps in reducing the memory footprint of a single VM. We perform an asymptotic cost analysis using KSM, which gives us the average cost of adding a container/VM on the host with similar guests. This helps us to study the impact of KSM on packing more guests on the same host, i.e., increasing the **density** of guests running on the same host infrastructure.

Experimental Details: We created VMs with 1 vCPU, 1 GB memory running Linux kernel 3.16 on a 3.16 host Linux system and containers with shared, copy-on-write chroot directory. For running multiple guests, we clone a base VM and container image respectively.

We start a guest and wait for the KSM numbers to stabilize and record the PSS value. Similarly for both–VMs and containers–we run a new guest, let the KSM numbers stabilize and record the total PSS for all running guests.

$$Average\ mem\ footprint = \frac{Total\ PSS\ of\ all\ running\ guests}{No\ of\ guests\ running} \tag{6.1}$$

Figure 6.2 compares the average memory cost of starting a new VM to the average cost of starting a container. It reflects the numbers with KSM feature enabled for the host and `qemu-kvm` configured to be KSM enabled. This graph measures the cost of an idle VM waiting for a login, in order to get a sense of the baseline costs of each approach. We spot-checked the confidence intervals for memory footprint of 4, 8, 16, 32 guests and the variance is negligible.

In the single guest case, the memory footprint of a container is around 16 MB. In contrast, the memory footprint of a VM is 187 MB—12× that of a container. This measurement implies that 12 times more containers than VMs can be supported on a given host with a fixed amount of memory.

With memory deduplication roughly 97 MB of each VM image can be reused as additional VMs are added. Asymptotically, the effective memory footprint of a VM is roughly 90 MB. Moreover, the average memory footprint per VM drops below 100 MB with as few as 7 VMs. As Linux containers benefit by sharing libraries and other files among processes through the host page cache their effective memory footprint is roughly 9 MB, asymptotically. Thus, simple memory deduplication helps both containers and VMs, but disproportionately improves the memory overhead of VM from 12× to 10× that of containers.

Comparison to Jails: As a point of comparison, we also measured the memory footprint of a FreeBSD jail, at 22 MB using RSS, as FreeBSD does not report a PSS. Although 22 MB is higher than the 16 MB measured for LXC, we expect this difference would be smaller if the PSS were measured in both cases. Either way, these numbers are comparable to a first order, adding evidence that our data is roughly representative of other container implementations.

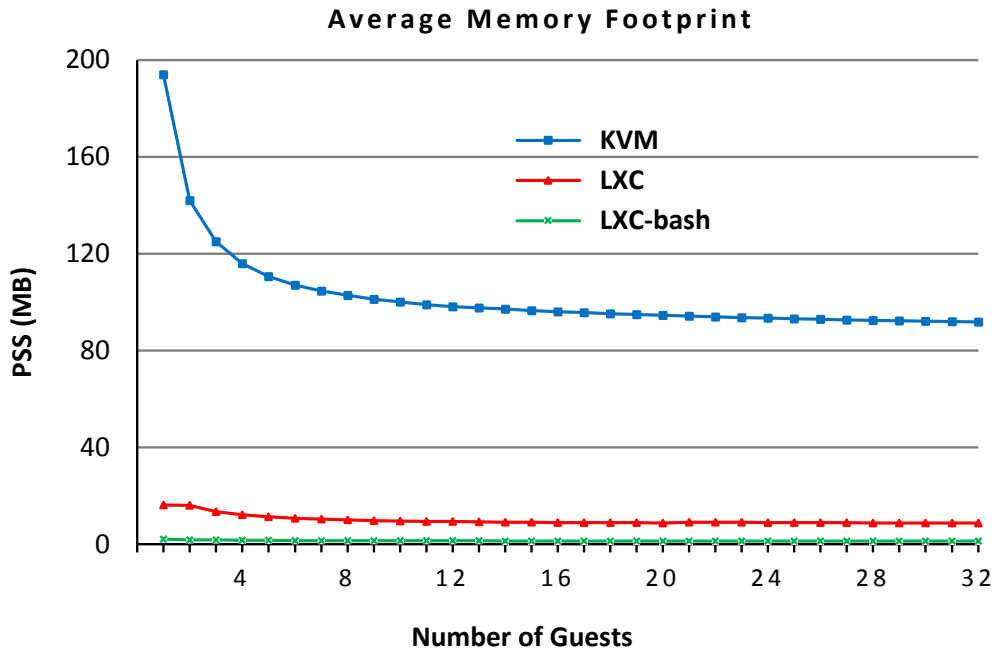

Figure 6.2: Average memory footprint (Proportional Set Size) of VMs with KSM enabled versus full Containers and application containers, as the number of instances increases. Lower is better.

Shared vs. Copied chroot: The experiments above are designed to show containers in their best light: with a shared, copy-on-write chroot directory. By sharing files, the memory footprint of the container drops by roughly half. In contrast, if one set up a *copy* of the chroot environment with identical contents, neither KSM nor the host page cache will deduplicate these contents. When we measured the asymptotic memory overheads in this case, containers were around 15.5 MB (vs 9 MB above). The only file sharing is in the LXC host process, which operates outside of the chroot directory. Without deduplication support for in-memory file contents, this common deployment scenario for containers will see higher overheads.

Single-Application Container: Figure 6.2 also shows the memory footprint of a single application, `/bin/bash` running in a sandbox. The memory footprint of a single-application container is 2 MB, compared to 16 MB for a full system container. Once file sharing is factored in among multiple identical application containers, the footprint drops to 1.3 MB in the limit. Although this is an unfair comparison in some respects, the single-application case is where containers are most useful, reducing memory footprint by up to 60× compared to dedicating a VM to a single application.

Comparison with Processes: Finally, we note that both VMs and containers have significantly higher memory overheads than a Linux process. As a point of comparison,

the simplest, "hello world" process on Linux is roughly 350 KB—two orders of magnitude less than a container. In contrast, a library OS, which runs a lightweight guest in a highly restricted picoprocess, can run "hello world" in 1.4 MB [47]—a factor of 6.4 lower than a shared chroot container. Thus, without much more aggressive optimization, if not an architectural shift, neither VMs nor containers will match the memory footprint of a simple OS process.

To summarize the impact of KSM on density, with the shared chroot optimization for containers, the average memory footprint gap between VMs and containers is 10x. However, if copied chroot containers were taken in account, the average memory footprint gap would be 5×—90MB effective memory footprint of VMs and 16 MB effective memory footprint of containers. However, since this work considers the best case for both–VMs and containers, the remaining work focuses on reducing the 10× gap between the effective memory footprint of VMs and containers.

6.5 Incremental Cost Analysis

Reducing the incremental cost of a VM can decrease the gap between the average memory footprint of containers and VMs. Incremental cost, is the amount of physical memory required to add a guest when a guest with same configuration already exists on a given physical host as shown in Figure 6.3. To account for the incremental cost of a VM, we analyze the physical pages mapped in the address space of both the VMs and count only the unique physical pages, which get added because of addition of a new VM.

Experimental Details: To determine the incremental cost of a VM, add a VM, $VM2$ with similar configuration as the existing guest VM, $VM1$ which is already running and let the KSM numbers stabilize. We account for the physical pages mapped in $VM1$ and $VM2$ from the algorithms stated in basic framework section and count the unique physical pages added to the address space of $VM2$. These unique pages give us the incremental cost of adding $VM2$.

As the Figure 6.3 shows, the incremental cost of adding $VM2$ is 91 MB. To understand how this incremental cost is being used, we analyze the unique physical pages added by addition of a VM and figure out the region to which they belong: guest RAM or QEMU. From the host perspective, 23 MB of this incremental cost constitutes the unique physical pages which get added because of QEMU device emulation and QEMU process's stack and heap. Remaining 68 MB is attributed to the guest RAM or the guest VMA allocated by QEMU process.

The pages that belong to guest RAM are further analyzed from within the guest, by getting the reverse EPT mappings and guest physical addresses corresponding to the uniquely added new host physical pages. Out of 68 MB of guest RAM incremental cost, 55 MB is guest file backed, 11 MB is guest anonymous and 2 MB is not mapped in the EPTs as shown in Figure 6.4.

To reduce this incremental cost, the cost from either emulated devices or the guest RAM should be reduced. We further look into memory reduction opportunities, which

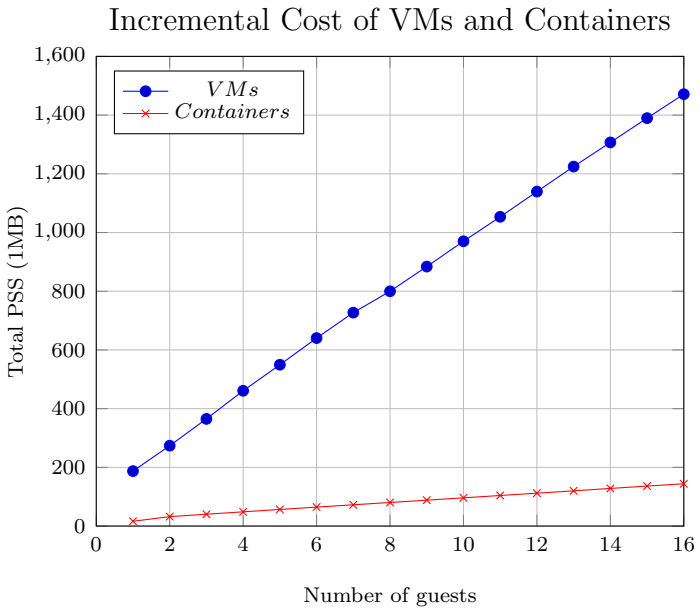

Figure 6.3: Incremental cost of VMs and containers. Incremental cost of a VM is 91 MB.

help us to bring down this incremental cost.

6.6 Memory Reduction Opportunities

This thesis, suggests three possible opportunities to reduce the incremental cost of a VM. First, to improve the existing deduplication techniques. Second, removal of unused devices emulated by QEMU. Third, removal of unused pages from the guest address space.

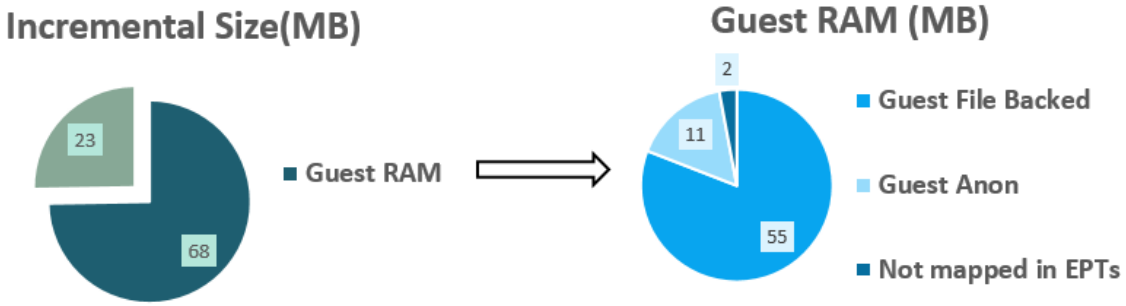

Figure 6.4: Incremental cost analysis

	Anon/Anon	File/File	Anon/File	Total
Within the VM process	28 MB	0.5 MB	0.5MB	**29 MB**
Between VM and Host	18 MB	2 MB	12 MB	**32 MB**
Between 2 VM processes	48 MB	8 MB	3 MB	**59 MB**
Total	**94 MB**	**10.5 MB**	**15.5 MB**	**120 MB**

Table 6.4: VM Memory footprint deduplication opportunities. All the guest RAM is considered anonymous memory. KSM can deduplicate column 1, but not columns 2 and 3. Column 2 identifies duplicates between QEMU-internal file use and the rest of the host. Column 3 identifies duplicate content between either guest RAM or QEMU anonymous memory, and files in the host.

6.6.1 Deduplication Opportunities

KSM helps to pack more guests on to the same host. To study the effectiveness of the deduplication technique, we analyze the total deduplication opportunities available for a VM and how much of it is actually being explored by KSM.

For this analysis, we disable huge pages and KSM in the host as well as guest. We calculate deduplication opportunities within a guest, between the host and a guest, and between two guests.

We hashed the contents of each physical page on the system and compared these hashes to the hashes of physical page frames assigned to one guest. We find duplicates of pages mapped to the VM process against the pages mapped to the same VM process, all the host pages and the pages mapped to another VM process irrespective of whether those pages are file-backed or anonymous. We found 120 MB of the VM's pages are duplicates, either with the host or another guest. Therefore, a perfect KSM could deduplicate 120 MB. Out of these 120 MB of duplicate pages, we found 29 MB duplicate pages within the VM process; 32 MB duplicate pages between the VM process and the host; and 59 MB duplicate pages between 2 VM processes (each row in Table 6.4).

KSM is designed to deduplicate the anonymous pages for the VM process against the VM itself (28 MB), anonymous pages of host (18 MB), and other VM processes (48 MB)—Column 1 of Table 6.4.

Thus, KSM at best can deduplicate 94 MB out of the 120 MB opportunity, missing another 26 MB that could be deduplicated. The second column indicates opportunities where files used by the QEMU process have duplicate contents with other files on the system *and are in distinct page frames and files*. The third column indicates opportunities where the contents of guest memory or QEMU anonymous memory are duplicated with a cached file on the host, such as the same version of `libc` in two places. If KSM deduplicates not just anonymous pages with other anonymous pages but also file-backed pages with other file-backed pages, we can deduplicate a total of 10.5 MB more pages — 2 MB between the VM and the host, 0.5 MB within the VM, and 8 MB between

2 VMs. These duplicate pages belong to some file that is opened by the VM process as well as the host or other VM. Similarly, if KSM deduplicates anonymous pages with file-backed pages, we can save the remaining 15.5 MB more — 12 MB between the VM and the host, 0.5 MB within the VM, and 3 MB between 2 VMs. These duplicate pages belong to say a shared library in the guest kernel read from the virtual disk and shared library in the host. The deduplication opportunities are shown in detail in Table 6.4.

In summary, we see a total of 26 MB additional duplicate pages that could be removed from the current 91 MB incremental cost of starting another VM, if KSM deduplication were able to deduplicate VM memory with files in the host page cache.

Many researchers [27,38,39,45] have explored the effectiveness of different strategies to find memory duplicates. Singleton [45] improves the performance of KSM in terms of time taken to find duplicate pages by replacing Red-Black tree implementation with hash tables. They also reduce memory footprint by evicting potentially double-cached, duplicate disk image data from the host page cache. XLH [39] deduplicates virtual disk I/O across VMs. Neither of these strategies, nor KSM, considers the case of two different files with the same contents, in active use by both a guest and host.

Thus, if file-backed vs. anonymous and file-backed vs. file-backed deduplication is supported by KSM, we can save an additional 26 MB from the effective memory footprint of a VM. This might come with a higher performance cost because a huge portion of memory might end up being advised as mergeable and being scanned in each pass. However, there has to be a trade off between performance and the amount of memory saved which complies with the service level objectives (SLOs). The additional deduplication potential found, accounts for a major portion of incremental cost, therefore, by improving the existing deduplication techniques we can save 26 MB out of 91 MB incremental cost thus bringing down the incremental cost to 65 MB.

6.6.2 Extra Devices

Another significant source of VM memory overhead is emulating unnecessary devices in QEMU. Looking into the possible avenues of reducing the memory footprint of a VM, we also consider devices that are emulated by QEMU and attached by default to the VMs created via virt-manager (a desktop driven virtual machine manager through which users can manage their virtual machines). One common scenario in cloud deployments is running a VM which just requires a terminal and no graphics or audio/video support. Such VMs function by just having a network device attached to them. In such a scenario, to make VMs as close as possible to containers without losing usability, we can safely remove the audio, video, and extra USB devices from a default KVM instance (as configured by the common `virsh` tool) without loss of important functionality.

Experimental Details: We run a VM with the standard devices configuration, with KSM enabled and note the PSS after the KSM numbers stabilize. Similarly, for each VM with custom configuration (one or more unused devices removed) we note its PSS after KSM stabilizes. To get the effective savings as shown in Table 6.5, we subtract the new custom VM PSS from the base VM PSS.

Device	Savings (in MB)
Video and Graphics	13.5
Audio	5.3
2 USB controllers	5.2
Video, Graphics and Audio	15.3
Video, Graphics and USB	15.2
Audio and USB	5.12
Video, Graphics, Audio and USB	13.6

Table 6.5: VM Memory footprint savings when disabling unused devices

Table 6.5 shows the effective savings in VM memory footprint when we disable various peripheral emulation. We note that though these combinations are not strictly additive but removing video, graphics and audio devices gives maximum possible savings, which is around 15.3 MB. We also observe that video and graphics devices tend to constitute a good portion of a VM memory footprint and in scenarios where VMs are just used to ssh and run workloads and do not require a graphical console, we can save on an average 13.5 MB and eventually provision more guests.

Thus this shows that there are opportunities apart from the duplicates, which if explored well can lead to a considerable reduction in memory footprint of a VM. We also suggest that the VMs should by default be provisioned with a minimal device configuration, which can be modified as per the user needs at run-time. This would help to save the extraneous memory wasted in large cloud deployments where users just SSH or login to the terminal of virtual machines.

Impact on incremental cost: Considering the best case for VMs that results in maximum memory savings, i.e., by removing video, graphics and audio devices, we measured its contribution towards the incremental cost of a VM. We configure the VMs with no video, graphics and audio devices and perform an average memory footprint analysis. We observe that out of 15.3 MB, 5.3 MB of savings constitutes a reduction in incremental cost whereas the remaining 10 MB eventually gets deduplicated as more VMs are added. Thus, for a singleton VM case, the savings by removal of unused devices is 15.3 MB whereas when packing more guests with similar configuration, effective savings are 5.3 MB. Thus, the incremental cost of a VM can further be reduced from 65 MB (after improving the deduplication techniques) to 60 MB by removing the unused devices.

6.6.3 Paravirtualized Memory Policies

Another source to reduce the memory footprint of a VM is to return the non-deduplicated free pages from guest kernel back to host. During boot time, a lot of unique pages are read by the guest VM and are freed right away. Moreover, whenever a page is freed in the guest in the course of its lifetime, it is put on the guest free list and never reclaimed back by the host unless the host is under memory pressure. As a result, there are few pages that are mapped in the VM process but are not really used by the guest. We can implement a paravirtualized memory policy, so that some of the pages on the guest

kernel free list that are not deduplicated with other pages are periodically kicked back by the guest to the host, which in turn removes those pages from the EPT tables for the guest and unmaps those pages from the VM process. Such a paravirtualized memory policy can help further reduce the memory footprint of a VM.

To calculate the amount of pages that are not being used by the guest and can be potential candidates for giving back to host, we consider the pages that are in the inactive LRU list of the guest. We refer to these pages as the bloat up because of their state — as they are not recently used — and are target of such policy implementation.

Such a page which is inactive and is in LRU list can have either the referenced bit set or unset. If the referenced bit is not set then it means that the page has not been accessed recently and is an idle candidate to be removed and given back to host. However, for the pages in LRU inactive list which have their referenced bit set, there might be two possible reasons for this state. First, since it takes two accesses for a page to be declared active, after the first access, referenced=1, but since it would be moved to active list on next access, active flag is still 0. Second, since it also takes two accesses for a page to be declared inactive, it can transition into a less active state where active=0 and referenced=1 and eventually after a timeout move to active=0 and referenced=0.

Considering the above scenarios, an active=0 but referenced=1 bit set page can also be a potential candidate for giving back to the host. However since such pages might soon be moved to active list, for a clear bloat up measurement we disregard such pages. Thus, counting the pages that are in LRU list and have active and referenced bit set as 0, gives us a lower bound to the memory savings possible by implementing such a paravirtualized memory policy.

Experimental Setup: We consider a 3.16 kernel host and guest with no KSM running. We pause the guest after it has run for some time and figure out all the physical addresses and their hashes for the entire system; physical page information for all the pages mapped in the guest, by using algorithms mentioned under basic framework.

To figure out the guest page flags, we get the guest physical addresses from the host physical addresses considering the EPT reverse mappings, and then use the guest physical addresses to find the page flag bits by reading /proc/kpageflags.

The pages that have LRU bit as 1, active bit as 0 and referenced bit as 0 constitute the base set for bloat up. We get the host physical addresses corresponding to these pages from the EPT mappings and get the hashes corresponding to them from the HostPhysicalPageInfo procedure described under basic framework. Now we remove the pages from the base bloat up set which have duplicate hashes with the remaining of the guest memory. This is done to remove those pages that might get deduplicated and be removed from the bloat up considering all the deduped opportunities are explored. The remaining pages, which have unique hashes in the bloat up set, are the ones which can be removed and can be target of this paravirtualized policy.

Our results show a savings of 5.38 MB for LRU, inactive pages with referenced bit unset. These numbers are for an idle VM with no workload running inside it. However, in scenarios where a workload has already run, there might be files, which are lying in the LRU inactive list and have not been removed for a long time. Such pages constitute

the bloat up and would bring down the guest memory footprint considerably when such a paravirtualized policy is implemented.

Impact on incremental cost: To study the impact of bloat up on incremental cost, we run two guests and analyze the unique physical pages added to $VM2$ from within the guest to determine the bloat up. We note that the bloat up constitutes nearly 2 MB of the incremental cost. Thus reducing the incremental cost further to 58 MB.

6.7 Deduplicating Different Guest Kernels

Throughout this thesis, we have looked into deduplication potential between the guests and hosts having the same kernel. The effectiveness of KSM, as seen till now, was under the assumption that all the VMs are running the same kernel thus making the environment equivalent to a container as all containers share the same host kernel. The experiment in Figure 6.2 shows the best case for KSM—the exact same kernel version in all VMs and the host. In this subsection, we measure the impact on deduplication when different versions of Linux are loaded in the guest than the host. In practice, even initially identical guests can diverge over time such as when one VM installs a security upgrade and one does not. We expect similar trends would hold among sets of guests. With a 3.16 host kernel, a VM running 3.17 kernel will deduplicate only 33 MB. If the VMs load a 3.15 kernel, KSM only finds 26 MB duplicates. Moreover, if we ran 2.5 year older 3.2 kernel in the VMs, KSM still finds 23 MB to deduplicate.

Although it is unsurprising that deduplication opportunities are inversely proportional to the differences in the kernel version, it is encouraging that a significant savings remain even among fairly different versions of the Linux kernel.

6.8 Summary

Thus, this Chapter provides careful analysis of the memory usage by VMs and containers. We note that PSS is a better metric to measure the memory footprint of a process as compared to RSS. KSM, memory deduplication technique facilitates provisioning more guests on the same host with fixed amount of memory. We conduct analysis to observe the effect of KSM on memory footprint of VMs and thereby consolidation density. Asymptotically, 10 times more containers can run on a host as compared to VMs using KSM. However, reducing the incremental cost of VMs can reduce this gap. We find three possible directions to reduce the 91 MB incremental cost; 26 MB by deduplicating file-backed vs. file-backed and file-backed vs. anonymous memory which is missed by KSM; 5.3 MB by removing unused devices like video, graphics and audio and another 2 MB by removing the unused pages from the guest address space; thus reducing the overall incremental cost from 91 to 58 MB.

As part of the future work, an in depth analysis of the use of guest file-backed incremental cost of 55 MB can help reduce the incremental cost further. As this thesis focuses on the user or guest-level memory utilization which dominates the memory cost,

we leave for future work a precise accounting of host kernel data structures dedicated to a VM or container. We expect containers impose more relative costs on the host kernel than a VM, as most emulation and other supporting data structures for KVM (beyond nested page tables) are in a second, user-level QEMU process, which is included in this analysis.

Chapter 7

Conclusion

VMs and containers are two fundamental virtualization technologies for deploying guests in data centres. Though VMs are supposed to have high overheads, this thesis shows that the gap between the VMs and containers is not as high as purported. We evaluate two critical factors for a cloud provider, latency and density, with respect to VMs and containers. The start up time for VMs is 2 orders of magnitude higher than that of containers but the start-up time can be reduced by an order of magnitude using checkpoint-restore. The effect of packing more guests on CPU bound workload is the same on containers as well as VMs, and the absolute performance of VMs and containers is equivalent for CPU-bound workload. VMs only suffer for I/O bound workloads more than Containers because of frequent VM exits which have been solved by research works like [26, 48].

As memory footprint determines the consolidation density, we study the metrics available for measuring memory footprint of VMs and containers and conclude that PSS gives more accurate measurements than RSS. We measure the effect of deduplication technique, KSM, on memory footprint of virtualization techniques and also measure the impact of KSM on density of VMs and containers. KSM, helps provisioning more KVM VMs but the same benefits are not offered to containers. Using this optimization, memory footprint of a VM is 5 times that of containers (using full clone VMs and containers) as compared to 14 times without any optimization. However, since this study considers the best case for containers, which is shared chroot directory, the effective memory footprint of containers is 9 MB as compared to 90 MB memory footprint of VMs. Thus, the actual asymptotic gap between VMs and containers is 10×. Thus to minimize this gap, we perform an incremental cost analysis on VMs and containers, and show that each VM requires extra 91 MB memory when added to a host with a same configuration VM running. Thus, to reduce the gap between VMs and containers, it is important to bring down this 91 MB incremental cost.

We suggest following three ways to reduce the gap between the memory footprint of VMs and containers. **First**, by improving existing deduplication opportunities, for which we analyze the deduplication potential between a guest and a host, a guest and a guest, and within a guest itself. When the total deduplication potential is 120 MB, KSM deduplicates only 94 MB leaving behind 26 MB, which can be deduplicated if KSM works on file-backed vs. file-backed and file-backed vs. anonymous memory. **Second**,

by removal of unused devices which are emulated by QEMU by default. The maximum reduction in memory footprint of a VM by removing such devices in singleton VM case is 15.3 MB. Removing video, graphics and audio devices can attain this memory reduction. However, the overall impact of removing devices on incremental cost is 5.3 MB as remaining savings are eventually deduplicated when more VMs with similar configuration are launched. **Third**, by removal of unused pages from the guest address space. This method helps to save 5.3 MB in a singleton VM case. However, this technique contributes only 2 MB in reducing the incremental cost. Thus, by above techniques, we can attain an overall reduction of 33 MB in the incremental cost, thereby reducing it from 91 MB to 58 MB.

Thus, both VMs and containers incur overheads and scalability bottlenecks. Depending on the critical resource, these bottlenecks may yield different overall consolidation densities. Our current measurements indicate that CPU-bound workloads are comparable either way, and I/O bound workloads are primarily sensitive to the multiplexing mechanism. Although the memory footprint and start-up times of containers tend to be lower, it is easy to craft an experiment that exaggerates the differences. With reasonable optimizations applied, start-up time of a VM can be reduced by an order of magnitude (6× instead of 50× of a full container) and the memory footprint of a VM can be reduced by a third.

7.1 Future Work

In future, we wish to evaluate the contribution of kernel data structures to the cost of VMs and containers. We expect containers to impose higher costs on host kernel as compared to VMs as they share the kernel address space with the host and most of emulation and data structures for a VM are accounted within the QEMU process itself. Thus, this might further bridge the gap between VMs and containers by raising the memory footprint of containers. This requires precise accounting and has been left as a future work.

Though, the pages which are not mapped in EPTs but are part of the guest VMA/RAM, contribute less towards the incremental cost, we still want to understand how they are being used. We analyzed the page flags corresponding to such pages but it did not help us much. Further analysis is required to answer the above question.

Looking from another perspective, to reduce the memory footprint of VMs, we aim to analyze the file backed guest memory, which contributes 55 MB towards the incremental cost and see how it is being used. This would help answer why such a significant amount of memory is not getting deduplicated and would further help us understand the portion of this cost that can be reduced by improving the deduplication technique.

Bibliography

[1] Clear Containers. https://clearlinux.org/features/clear-containers.

[2] CRIU. Online at http://www.criu.org/.

[3] FreeBSD Jails. https://www.freebsd.org/doc/handbook/jails.html.

[4] KSM Preload. http://vleu.net/ksm_preload/.

[5] KVM and Docker LXC Benchmarking with OpenStack. http://bodenr.blogspot.com/2014/05/kvm-and-docker-lxc-benchmarking-with.html.

[6] Linux container. https://linuxcontainers.org/.

[7] SBU CS Tech Day. https://www.cs.stonybrook.edu/about-us/News/Sept-12-Computer-Science-Technology-Day.

[8] The Rekall Memory Forensic framework. Online at http://www.rekall-forensic.com/.

[9] UKSM. http://kerneldedup.org/en/projects/uksm/introduction/.

[10] VBox Page Fusion. https://www.virtualbox.org/manual/ch04.html#guestadd-pagefusion.

[11] Xen Dedupe. http://kerneldedup.org/en/projects/xen-dedup/introduction/.

[12] K. Agarwal, B. Jain, and D. E. Porter. Containing the hype. In *Asia-Pacific Workshop on Systems*, 2015.

[13] O. Agesen, J. Mattson, R. Rugina, and J. Sheldon. Software techniques for avoiding hardware virtualization exits. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 35–35, Berkeley, CA, USA, 2012. USENIX Association.

[14] N. Amit, M. Ben-Yehuda, D. Tsafrir, and A. Schuster. viommu: Efficient iommu emulation. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'11, pages 6–6, Berkeley, CA, USA, 2011. USENIX Association.

[15] A. Arcangeli, I. Eidus, and C. Wright. Increasing memory density by using ksm. In *Proceedings of the linux symposium*, pages 19–28, 2009.

[16] S. K. Barker, T. Wood, P. J. Shenoy, and R. K. Sitaraman. An empirical study of memory sharing in virtual machines. In *USENIX Annual Technical Conference*, pages 273–284, 2012.

[17] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. Ix: A protected dataplane operating system for high throughput and low latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 49–65, Broomfield, CO, Oct. 2014. USENIX Association.

[18] N. Bila, E. J. Wright, E. D. Lara, K. Joshi, H. A. Lagar-Cavilla, E. Park, A. Goel, M. Hiltunen, and M. Satyanarayanan. Energy-oriented partial desktop virtual machine migration. *ACM Trans. Comput. Syst.*, 33(1):2:1–2:51, Mar. 2015.

[19] E. Bugnion, S. Devine, K. Govil, and M. Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 15(4):412–447, 1997.

[20] B. Calkins. *Oracle Solaris 11 System Administration*. Prentice Hall Press, Upper Saddle River, NJ, USA, 1st edition, 2013.

[21] Canonical. LXD crushes KVM in density and speed. `https://insights.ubuntu.com/2015/05/18/lxd-crushes-kvm-in-density-and-speed/`, 2015.

[22] L. Chen, Z. Wei, Z. Cui, M. Chen, H. Pan, and Y. Bao. Cmd: classification-based memory deduplication through page access characteristics. In *Proceedings of the 10th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 65–76. ACM, 2014.

[23] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio. An updated performance comparison of virtual machines and linux containers. Technical Report RC25482(AUS1407-001), IBM Research Division, 11501 Burnet Road, Austin, TX, 2014.

[24] Filebench. `http://sourceforge.net/projects/filebench/`.

[25] B. Ford and R. Cox. Vx32: Lightweight user-level sandboxing on the x86. In *Proceedings of the USENIX Annual Technical Conference*, pages 293–306, 2008.

[26] A. Gordon, N. Amit, N. Har'El, M. Ben-Yehuda, A. Landau, A. Schuster, and D. Tsafrir. Eli: Bare-metal performance for i/o virtualization. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 411–422, New York, NY, USA, 2012. ACM.

[27] D. Gupta, S. Lee, M. Vrable, S. Savage, A. C. Snoeren, G. Varghese, G. M. Voelker, and A. Vahdat. Difference engine: Harnessing memory redundancy in virtual machines. *Communications of the ACM*, 53(10):85–93, 2010.

[28] J. L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, 2006.

[29] W. Jian, J. C.-F. Du Wei, and X. Xiang-Hua. Overcommitting memory by initiative share from kvm guests. *International Journal of Grid & Distributed Computing*, 7(4), 2014.

[30] R. Kapoor, G. Porter, M. Tewari, G. M. Voelker, and A. Vahdat. Chronos: Predictable low latency for data center applications. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 9:1–9:14, New York, NY, USA, 2012. ACM.

[31] Kernel-based virtual machine. `http://www.linux-kvm.org/`.

[32] J. F. Kloster, J. Kristensen, and A. Mejlholm. Determining the use of interdomain shareable pages using kernel introspection. Technical report, Technical report, Aalborg University, 2007.

[33] H. A. Lagar-Cavilla, J. A. Whitney, A. M. Scannell, P. Patchin, S. M. Rumble, E. De Lara, M. Brudno, and M. Satyanarayanan. Snowflock: rapid virtual machine cloning for cloud computing. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 1–12. ACM, 2009.

[34] J. Liu, W. Huang, B. Abali, and D. K. Panda. High performance vmm-bypass i/o in virtual machines. In *Proceedings of the Annual Conference on USENIX '06 Annual Technical Conference*, ATEC '06, pages 3–3, Berkeley, CA, USA, 2006. USENIX Association.

[35] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft. Unikernels: Library operating systems for the cloud. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[36] J. N. Matthews, W. Hu, M. Hapuarachchi, T. Deshane, D. Dimatos, G. Hamilton, M. McCabe, and J. Owens. Quantifying the performance isolation properties of virtualization systems. In *Proceedings of the 2007 Workshop on Experimental Computer Science*, ExpCS '07, New York, NY, USA, 2007. ACM.

[37] D. Meisner, J. Wu, and T. Wenisch. Bighouse: A simulation infrastructure for data center systems. In *Performance Analysis of Systems and Software (ISPASS), 2012 IEEE International Symposium on*, pages 35–45, April 2012.

[38] K. Miller, F. Franz, T. Groeninger, M. Rittinghaus, M. Hillenbrand, and F. Bellosa. Ksm++: Using i/o-based hints to make memory-deduplication scanners more efficient. In *Proceedings of the ASPLOS Workshop on Runtime Environments, Systems, Layering and Virtualized Environments (RESoLVE'12)*, 2012.

[39] K. Miller, F. Franz, M. Rittinghaus, M. Hillenbrand, and F. Bellosa. Xlh: More effective memory deduplication scanners through cross-layer hints. In *USENIX Annual Technical Conference*, pages 279–290, 2013.

[40] D. G. Murray, H. Steven, and M. A. Fetterman. Satori: Enlightened page sharing. In *In Proceedings of the USENIX Annual Technical Conference*. Citeseer, 2009.

[41] NIST. National Vulnerability Database. `http://nvd.nist.gov/`, 2008.

[42] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe. Arrakis: The operating system is the control plane. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 1–16, Broomfield, CO, Oct. 2014. USENIX Association.

[43] D. E. Porter, S. Boyd-Wickizer, J. Howell, R. Olinsky, and G. Hunt. Rethinking the library OS from the top down. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 291–304, 2011.

[44] N. Regola and J.-C. Ducom. Recommendations for virtualization technologies in high performance computing. In *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pages 409–416, Nov 2010.

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

[46] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson. Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 275–287, New York, NY, USA, 2007. ACM.

[47] C.-C. Tsai, K. S. Arora, N. Bandi, B. Jain, W. Jannen, J. John, H. A. Kalodner, V. Kulkarni, D. Oliveira, and D. E. Porter. Cooperation and Security Isolation of Library OSes for Multi-Process Applications. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, pages 9:1–9:14, 2014.

[48] C.-C. Tu, M. Ferdman, C.-t. Lee, and T.-c. Chiueh. A comprehensive implementation and evaluation of direct interrupt delivery. In *11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environment (VEE)*, VEE '15. ACM, 2015. recognized as Best Paper by the program committee.

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

[50] M. G. Xavier, M. V. Neves, F. D. Rossi, T. C. Ferreto, T. Lange, and C. A. F. De Rose. Performance evaluation of container-based virtualization for high performance computing environments. In *Proceedings of the 2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, PDP '13, pages 233–240, Washington, DC, USA, 2013. IEEE Computer Society.

[51] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*, 2009.