Containing the Hype

Kavita Agarwal, Bhushan Jain, and Donald E. Porter
Stony Brook University
{*kaagarwal, bpjain, porter*}@*cs.stonybrook.edu*

Abstract

Containers, or OS-based virtualization, have seen a recent resurgence in deployment. The term "container" is nearly synonymous with "lightweight virtualization", despite a remarkable dearth of careful measurements supporting this notion. This paper contributes comparative measurements and analysis of both containers and hardware virtual machines where the functionality of both technologies intersects. This paper focuses on two important issues for cloud computing: density (guests per physical host) and start-up latency (for responding to load spikes). We conclude that the overall density is highly dependent on the most demanded resource. In many dimensions there are no significant differences, and in other dimensions VMs have significantly higher overheads. A particular contribution is the first detailed analysis of the biggest difference—memory footprint—and opportunities to significantly reduce this overhead.

1. Introduction

Operating System-level virtualization, also known as a *container*, is an increasingly popular approach to isolating applications that use the same underlying OS kernel [7, 34, 37, 38]. Containers have recently gained popularity as the default back-end for Docker, an application packaging and distribution system used by companies including Google [26].

The purported reason to use containers over a hardware virtual machine, such as VMware or Xen, is reduced overheads. Containers forego the ability to run different OSes—an essential feature of VMs, but can be appropriate for scenarios where all guest applications are programmed to the same OS API. Containers are implemented by copying a subset of OS data structures, which one would expect to be lighter-weight than running another complete OS instance. Similarly, data structure initialization can be faster than booting a legacy OS kernel.

This difference in implementation techniques 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 [31] 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.

Moreover, there is a remarkable lack of scientific studies on how much performance benefits containers actually offer in exchange for these qualitative differences, and existing studies are either dated in a rapidly evolving field; narrowly focused on particular application areas; or overlook important optimizations, leading to exaggerated results. In the interest of adding experimental data to the scientific discourse on this topic, this paper contributes a careful, updated comparison of the two technologies.

Throughout the paper, we use Linux's KVM [18] and LXC [2] 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

APSys '15, July 27–28, 2015, Tokyo, Japan.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3554-6/15/07…$15.00.
DOI: http://dx.doi.org/10.1145/2797022.2797029

a first-order effect on memory overhead and is comparable to other VMs such as Xen, VMware, and VirtualBox. The design of LXC is also similar to other container implementations: FreeBSD Jails are implemented by allocating a separate prison object for the Jailed process [24], and Solaris Zones are implemented by configuring separate filesystem and virtual network interfaces for each zone [23]. As a first step towards empirically validating that LXC is representative of other container implementations, we compare the memory footprint and start-up time of LXC containers with FreeBSD Jails, which is within the same order of magnitude as LXC. An exhaustive comparison of each VM and container implementation is beyond the scope of this paper.

We organize this comparison around two important metrics for cloud efficiency: consolidation *density*, or guests per physical machine, and the latency to start a new guest—a metric relevant to quickly servicing spikes in demand. If start-up latency is too high, providers will generally provision for peak demand instead of average demand in order to minimize worst-case request latency.

Our comparison yields a more nuanced set of benefits and some drawbacks to each option. For instance, simple checkpointing optimizations yield start-up times for KVM that are an order of magnitude lower than reported by previous studies [11]. Moreover, the start-up time of either is unacceptably high to dynamically scale instances of latency-sensitive applications. Similarly, overall density of either technology is highly dependent on the most contended resource, and, for several resources, neither approach has a clear advantage.

Finally, this paper contributes an analysis of the worst case for VMs relative to containers: memory consumption. We identify several opportunities to improve both technologies. We expect that the limited functionality and security concerns for containers are fundamental to the design, and thus believe creating VMs with overheads equivalent to containers is a useful and interesting "challenge problem" for future research. We expect this challenge may not be completely realizable, as there are likely fundamental memory overheads for VMs relative to containers, but we also identify considerable bloat that could be reduced.

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 VM running the same guest OS, this paper primarily measures this comparison, although some data is also reported for the "single" case, as the "single" configuration is also a popular option. In general, we feel the "single" case is more fairly compared to a library OS [21, 33, 39] or a sandboxing system [13, 43].

Experimental Setup. We ran all experiments 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. Our host system runs Ubuntu 14.10 server with host Linux kernel version 3.16. Our host system includes KSM and KVM on QEMU version 2.1.0 and LXC version 1.1.0. Each KVM Guest is deployed with 1 virtual CPU with extended page tables (EPT) confined to a single host CPU core, 1 GB RAM, a 20 GB 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).

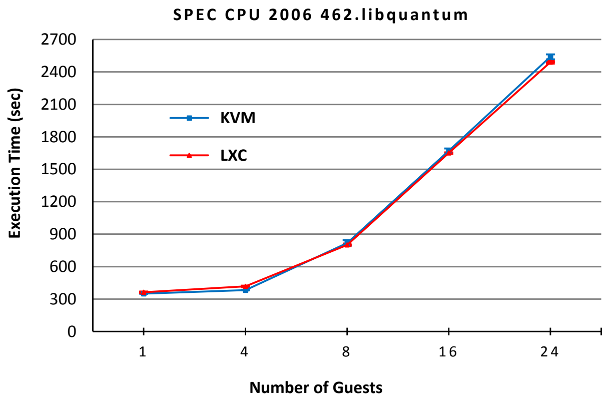

Figure 1. SPEC CPU 2006 execution time for simulating a quantum computer as number of guests increases on a 4-core machine. Lower is better.

2. Density of CPU and I/O Bound Workloads

In this section we measure the impact of increasing numbers of guests on a shared physical host. We measure the change in performance of one guest on a CPU-bound and I/O-bound workload as the number of guests per core increases.

2.1 CPU-Bound Workloads

We measure the performance of two SpecCPU 2006 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 run the benchmark on one guest, and 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).

Figures 1 and 2 show the execution time of the quantum and chess benchmarks as guest density increases, where lower is better. With only one guest on the system, perfor-

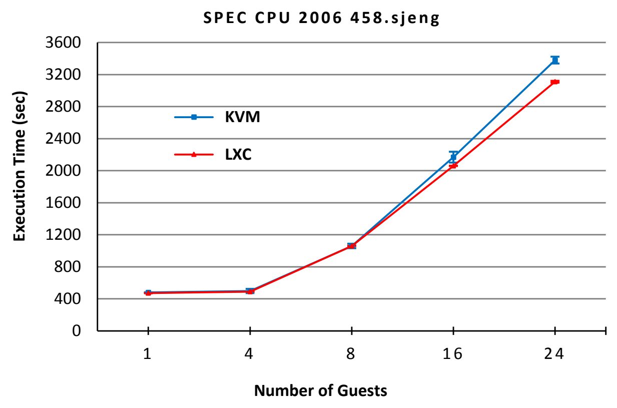

Figure 2. SPEC CPU 2006 execution time for chess simulation as number of guests increases on a 4-core machine. Lower is better.

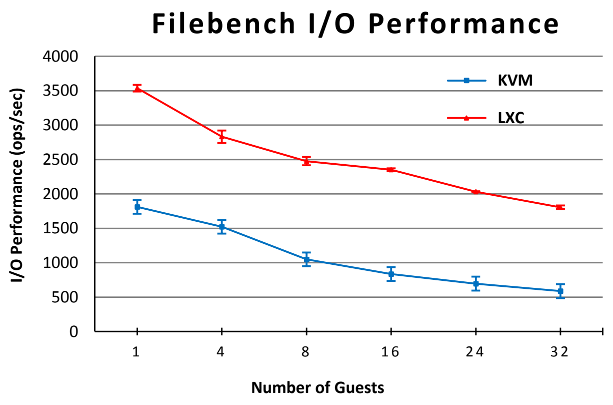

Figure 3. Filebench I/O performance as number of guests increases on a 4-core machine. Higher is better.

mance on both a VM and container are comparable to a native process. As density increases, both VMs and containers degrade dramatically and 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 appear to be equally good.

2.2 I/O-Bound Workloads

We use Filebench [12] to measure the performance of a typical fileserver workload running in one guest. The fileserver workload measures performance of various file operations like stat, open, read, 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 of incrementing an integer. Here, we measure the effect of CPU-bound workload running on other guests, over the I/O performance measured by Filebench. Figure 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. We believe this difference is attributable to the cost of VM exits incurred every few I/Os when using virtio, paravirtualized device drivers, or doubled host and guest filesystem overheads.

We also evaluated the effect of running an I/O-bound background workload on 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. We are still investigating this issue as ongoing work. 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 comparable 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 eliminate VM exits and yield I/O performance comparable to bare metal [3, 4, 14, 20, 40]. Moreover, recent papers indicate that the best performance is likely achieved when I/O drivers are pushed into the application [6, 32], 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.

3. Memory Overheads

The biggest difference in overheads between containers and VMs is their memory cost. We use the term **memory footprint** to refer to the incremental memory cost of adding another guest to a system. This study focuses on the user- or guest-level memory utilization, which dominates the memory cost. We leave a precise accounting of host kernel data structures dedicated to a VM or container for future work. We note that, for KVM, most supporting data structures (except the nested page tables) are in a second, user-level qemu process, which is included in this analysis.

To measure container memory footprint, we sum the proportional set size (PSS) of each process in the container. PSS accounts a portion of each shared page to each process, allowing correct sums across processes. For instance, if 10 processes in a container share 10 pages, each process's PSS is incremented by 1 page.

Figure 4 presents measurements comparing the incremental memory cost of starting a new VM, compared to the incremental cost of starting a container. 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. This graph measures the cost of

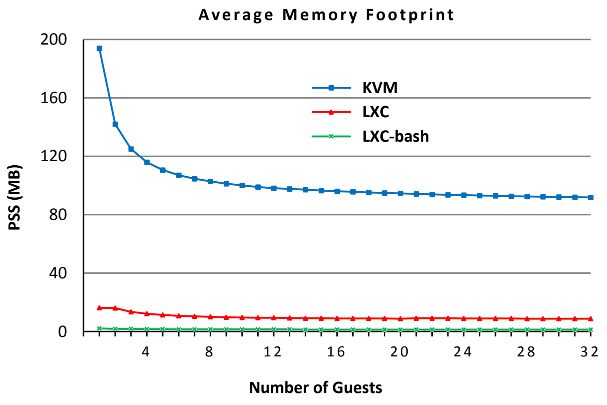

Figure 4. Average memory footprint (proportional set size) of VMs with KSM enabled versus full Containers and application containers, as the number of instances increases. Lower is better.

an idle VM waiting for a login, in order to get a sense of the baseline costs of each approach. After starting each new guest, we wait for the KSM numbers and memory consumption to stabilize before recording the memory footprint. We spot-checked the confidence intervals for memory footprint of 4, 8, 16, 32 guests and the variance is negligible.

When emulating a full Linux system, the footprint of a container is 16 MB. In contrast, the memory footprint of a VM is 194 MB—12× larger than a container in the same scenario. These measurements imply that 12 times more containers, than VMs can be supported on a given host with a fixed amount of memory.

However, a well-known technique for over-committing memory is hash-based deduplication [41], or Kernel Same Page Merging (KSM) on Linux. This technique hashes guest pages to identify duplicate contents, and then remaps the pages copy-on-write to a single copy. This is particularly effective when multiple instances of the same exact guest OS are running—i.e., normalizing to the constraint imposed by containers. Figure 4 reflects using the KSM feature for the host and configuring `qemu-kvm` to use the KSM feature.

With memory deduplication, roughly 96 MB of each VM image could be reused as additional VMs were added. In the asymptote, a VM's memory footprint is roughly 91 MB. With as few as 8 VMs, the average memory footprint drops below 100 MB. 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. Linux containers do not use KSM to deduplicate memory.

Linux containers primarily share libraries and other files among processes through the host page cache; the incremental cost for a container is roughly 8 MB in the limit. The average memory footprint per container drops below 10 MB with as few as 8 full containers. Simple memory deduplication helps both containers and VMs, but disproportionately improves the memory overhead of VM from 12× to 11×.

To determine whether KSM would afford additional benefits to containers beyond file-level sharing, we manually analyzed the anonymous pages between two container instances to find duplicates. We found only 1 duplicate anonymous page was among the processes of two containers, indicating KSM would afford little benefit to 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.

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 14 MB (vs 8 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 4 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.

Deduplicating Different Guest Kernel Versions The experiment in Figure 4 shows the best case for KSM—the exact same kernel version in all VMs and the host. This environment is equivalent to a container, as all containers share the same host kernel. Note that, even in the single-VM case, KSM can deduplicate memory with the host kernel, yielding a savings of 35 MB over KSM disabled.

Here we measure the impact on deduplication when different versions of Linux are loaded in the guest than is in 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

	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 1. VM Memory footprint deduplication opportunities. All 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.

3.15 kernel, KSM only deduplicates 26 MB. However, KSM still finds 23 MB to deduplicate between kernel versions 3.2 and 3.16—a gap of 2.5 years.

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

Discussion. This section demonstrates that content-based memory deduplication can significantly reduce the incremental cost of VMs relative to containers, and that the most fair comparison is in the asymptote, not the single-guest case. Even when the kernels are fairly different versions of Linux, the deduplication opportunities are significant. That said, containers still offer a memory footprint roughly $11-60\times$ lower than VMs, even with deduplication. As the next Section shows, other techniques can further narrow, but not close, this gap. Finally, the memory-saving benefits of containers are only realized when the system administrator is careful to share the same files copy-on-write, such as with a unioning file system [5]. These overheads could be mitigated if the kernel employed content-based deduplication for files cached in memory.

4. Memory Reduction Opportunities

This section proposes a challenge problem: Is it feasible to reduce the incremental memory cost of a VM to match a container? This section analyzes the memory usage of a VM and identifies opportunities to further reduce memory consumption without disrupting performance or functionality.

Figure 4 shows an asymptotic VM footprint around 91 MB. Among these 91 MB of unique pages, 23 MB belong to the userspace qemu emulator process, 2 MB are used for EPT tables, and 66 MB are used as RAM by the guest. The 23 MB belonging to qemu are used for device emulation, qemu's internal heap, and stack. Among the 66 MB used as guest RAM, 11 MB are allocated by the guest as anonymous memory (i.e., process heaps and stacks) and 55 MB pages are caching files in the guest.

The preliminary analysis below identifies two opportunities to further lower the asymptotic memory cost of a VM by 31.3 MB, to roughly 60 MB, or only $4-30\times$ higher than a container. Work is ongoing to identify additional opportunities and answer the larger challenge question.

4.1 Deduplication Opportunities

This subsection measures an upper bound on the reduction of VM memory via deduplication. For this analysis, we disable hugepages and KSM in the host as well as guest and quiesce the guests before analyzing memory pages. We hashed the contents of each physical page on the system and compared these hashes to the physical page frames assigned to one guest. We calculate deduplication opportunities within a guest, between the host and a guest, and between two guests. The breakdown by category of this analysis is shown in Table 1, and explained below.

We found 120 MB of the VM's 235 MB total pages are duplicates, either with the host or another guest. Therefore, a perfect KSM could deduplicate 120 MB. Each row in Table 1 shows how many duplicate pages can be eliminated by each possible comparison: 29 MB duplicate pages within the VM process itself, 32 MB duplicate pages between the VM process and the host, and 59 MB duplicate pages between 2 VM processes. All three categories are significant opportunities, although inter-guest deduplication is most promising.

KSM is designed to deduplicate only anonymous pages—Column 1 of Table 1. At best, KSM can deduplicate 94 MB out of the 120 MB opportunity, reducing the asymptotic incremental cost to 91 MB, and missing another 26 MB. The second column indicates opportunities where files used by the qemu process have contents duplicate to contents of 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. The deduplication opportunities are shown in detail in Table 1.

In summary, 26 MB of additional duplicate pages could be removed from the current 91 MB incremental VM cost if KSM were extended to files in the host page cache.

Many researchers [16, 28, 29, 36] have explored the effectiveness of different strategies to find memory duplicates. Sharma and Kulkarni [36] reduce the time for KSM to find duplicate pages by switching the KSM index from an RB-tree to a hash table. They also reduce memory footprint by evicting potentially double-cached disk image data from the host page cache. Miller et al. [29] deduplicate 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.

4.2 Extra Devices

Another significant source of VM memory overhead is emulating unnecessary devices in qemu. One common scenario

Device	Savings (in MB)
Video and Graphics	13.5
Audio	5.3
2 USB controllers	5.2

Table 2. VM Memory footprint savings when disabling unused devices.

is running a VM in a cloud that needs to only interact with a user over the network. In such a scenario, 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. Table 2 shows the effective savings in VM memory footprint when we disable various peripheral emulations. We note that these combinations are not strictly additive—removing video graphics and USB saves 15.3 MB, but removing all three saves only 13.6 MB. Work is ongoing to understand these diminishing returns. We see 5.3 MB reduction in the 91 MB incremental cost by not emulating unnecessary devices in addition to the memory deduplication opportunities.

5. Start-Up Latency

Another benefit of containers is the reduced start-up time compared to a VM. Lower start-up latency is of particular value to a cloud provider, as a sufficiently low startup latency allows the provider to provision for average load, rather than peak load, without missing a response deadline, such as those specified in a service level objective (SLO).

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. We measure this difference in Table 3. A typical VM takes 2 orders of magnitude longer to start up than a container.

Running a single-application container cuts the start-up time of LXC in half, but indicates a considerable startup 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.

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 the case of a VM, this elides the majority of these overheads, reducing start-up time by an order of magnitude. In contrast, for a container, the savings of checkpoint/restore are more marginal (0.008s for a full container and 0.027s for the application sandbox).

Even with the checkpoint/restore optimization, there is still a factor of $6\times$ difference in the start-up latency between the VM and a full container. However, there are likely opportunities to further close this gap, and we expect that optimizations to restore time will disproportionately help VMs.

Technique	time(sec)
KVM Start-Up	10.342
LXC Start-Up	0.200
LXC-bash Start-Up	0.099
FreeBSD Jail Start-Up	0.090
KVM Restore	1.192
LXC Restore	0.192
LXC-bash Restore	0.072

Table 3. Start-Up times for a VM and a container

For instance, the SnowFlock [19] VM Fork system reduces the start-up time of forked Xen VM by $3 - 4\times$ by only allocating and copying forked guest memory on demand. Bila et al. [8] show that the mean working set of a 4 GB VM is 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 difficult to precisely analyze 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 [17, 25]. For instance, in the BigHouse workload model, this would increase service times anywhere from a factor of $2\times$, 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 reported that a Linux process can start in 208 microseconds and Graphene itself can start in 641 microseconds [39].

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.

6. Related Work

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, and provides a useful, independent data for comparison.

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) [10]. Their density measurements show container density $14.5\times$ higher than KVM, limited by memory, whereas our measurements show the cost relative to LXC at $11\times$. 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.

Felter et al. [11] measure CPU, memory throughput, storage and networking performance of Docker and KVM. This study concludes that containers result in equal or better performance than VMs in almost all cases. 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 reports KVM's start-up time to be $50\times$ slower than Docker—commensurate with our unoptimized measurements; our measurements show that checkpoint and restore optimization to start a VM which brings KVM start-up time from $50\times$ to $6\times$. Another study measures the memory footprint of Docker and KVM using OpenStack [1], finding that KVM has $6\times$ larger memory footprint when running real application workloads. Our work contributes opportunities to reduce this $6\times$.

Regola and Ducom [35] 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 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 [3, 4, 14, 20]. Xavier et al. [42] 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 [22] 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. [37] measure relative scalability of Linux-VServer, another container technology, and Xen. They show that the Linux-Vserver performs $2\times$ 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.

A few research papers reduce the memory footprint of a VM or enhance RAM deduplication techniques to find more duplicates in less time [16, 30, 41]. Bugnion et al. [9] introduced transparent page sharing, where duplicate copies of a page are replaced by a reference to the copy-on-write mapped original page. Waldspurger [41] first implemented the inter-VM content based page sharing in VMware ESX server. ESX scans the VM's physical memory to find and deduplicate pages with duplicate content. Gupta et al. [16] extended the idea for Xen with a combination of whole page sharing, page patching and compression techniques. Murray et al. [30] also deduplicates pages in guests with sharing-aware virtual block devices in Xen. These devices detect sharing opportunities in the page cache immediately as data is read into memory. Miller [27] explains the reasons for memory duplication as popular pages like zero page, direct copies created by `memcpy, bcopy, and memmove` or deterministic results of the same program run multiple times simultaneously. He also analyses the spatial and temporal qualities of duplicate pages and sharing potential of 2 processes or VMs running the same workload. Groninger [15] also categorizes the duplicate pages of VMs based on memory access patterns and semantic information. This paper continues this line of work, investigating the fundamental opportunity for deduplication and exploring other avenues to further reduce the memory footprint.

7. Conclusion

VMs and containers provide some overlapping functionality—the ability to emulate a complete OS. However, these technologies were designed to address different goals and have significant qualitative differences: whether kernels can be different or the same; whether kernel modules can be loaded; whether a single application can run in isolation; the general ease of creation and use; and different security risks stemming from different attack surfaces and shared trusted computing base. This paper analyzes the density and start-up latency arguments for containers when both approaches overlap: running multiple instances of the same, complete OS. These results are preliminary: the community would benefit from further measurements of more applications, VM and container implementations, and other use cases.

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 full container is $6\times$ lower than a VM, and memory footprint is $11\times$ lower than a VM. On the other hand, the start-up time of an application container is $16\times$ lower than a VM, and memory footprint is $60\times$ lower than a VM. Our analysis identifies two simple opportunities to further reduce incremental VM memory usage by a third, we expect to discover others in future work.

8. Acknowledgments

We thank the anonymous reviewers, Eyal de Lara, Mike Ferdman, Ashvin Goel, Scott Stoller, and our shepherd Robbert van Renesse for insightful comments on this work. This research is supported in part by NSF grants CNS-1149229, CNS-1161541, CNS-1228839, CNS-1405641, and CNS-1408695. Bhushan Jain is supported by an IBM Ph.D. Fellowship.

References

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

[2] Linux Containers. `https://linuxcontainers.org/`.

[3] 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.

[4] 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*, USENIX ATC'11, pages 6–6, Berkeley, CA, USA, 2011. USENIX Association.

[5] Aufs. `http://aufs.sourceforge.net/`.

[6] 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.

[7] S. Bhattiprolu, E. W. Biederman, S. Hallyn, and D. Lezcano. Virtual servers and checkpoint/restart in mainstream Linux. *ACM SIGOPS Operating Systems Review*, 42:104–113, July 2008.

[8] 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.

[9] 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.

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

[11] 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.

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

[13] 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.

[14] 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.

[15] T. Gröninger. On Statistical Properties of Duplicate Memory Pages. Diploma thesis, System Architecture Group, Karlsruhe Institute of Technology (KIT), Germany, Oct.31 2013. `http://os.itec.kit.edu/`.

[16] 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.

[17] 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.

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

[19] 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.

[20] J. Liu, W. Huang, B. Abali, and D. K. Panda. High Performance VMM-bypass I/O in Virtual Machines. In *Proceedings of the Conference on USENIX '06 Annual Technical Conference*, ATEC '06, pages 3–3, Berkeley, CA, USA, 2006. USENIX Association.

[21] 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.

[22] 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.

[23] J. Mauro and R. McDougall. *Solaris Internals (2Nd Edition)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2006.

[24] M. K. McKusick and G. V. Neville-Neil. *The Design and Implementation of the FreeBSD Operating System*. Pearson Education, 2004.

[25] 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.

[26] C. Metz. Google Embraces Docker, the Next Big Thing in Cloud Computing. *WIRED*, June 2014. `http://www.wired.com/2014/06/eric-brewer-google-docker/`.

[27] K. Miller. *Efficient Main Memory Deduplication Through Cross Layer Integration*. PhD thesis, Karlsruhe, Karlsruher Institut für Technologie (KIT), Diss., 2014, 2014.

[28] 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.

[29] 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.

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

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

[32] 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.

[33] 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.

[34] D. Price and A. Tucker. Solaris Zones: Operating system support for consolidating commercial workloads. In *Proceedings of the Large Installation System Administration Conference (LISA)*, pages 241–254, 2004.

[35] 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.

[36] 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.

[37] 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.

[38] M. Stokely and C. Lee. The FreeBSD Handbook, 3rd Edition, Vol 1: Users's Guide, 2003.

[39] 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.

[40] 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.

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

[42] 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.

[43] 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.