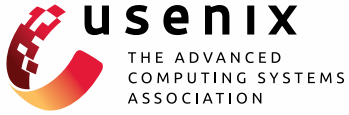

OSv—Optimizing the Operating System for Virtual Machines

Avi Kivity, Dor Laor, Glauber Costa, Pekka Enberg, Nadav Har'El, Don Marti, and Vlad Zolotarov, *Cloudius Systems*

https://www.usenix.org/conference/atc14/technical-sessions/presentation/kivity

This paper is included in the Proceedings of USENIX ATC '14: 2014 USENIX Annual Technical Conference.

June 19–20, 2014 • Philadelphia, PA

978-1-931971-10-2

Open access to the Proceedings of USENIX ATC '14: 2014 USENIX Annual Technical Conference is sponsored by USENIX.

OS^v— Optimizing the Operating System for Virtual Machines

Avi Kivity Dor Laor Glauber Costa Pekka Enberg
Nadav Har'El Don Marti Vlad Zolotarov
Cloudius Systems
{avi,dor,glommer,penberg,nyh,dmarti,vladz}@cloudius-systems.com

Abstract

Virtual machines in the cloud typically run existing general-purpose operating systems such as Linux. We notice that the cloud's hypervisor already provides some features, such as isolation and hardware abstraction, which are duplicated by traditional operating systems, and that this duplication comes at a cost.

We present the design and implementation of OS^v, a new guest operating system designed specifically for running a single application on a virtual machine in the cloud. It addresses the duplication issues by using a low-overhead library-OS-like design. It runs existing applications written for Linux, as well as new applications written for OS^v. We demonstrate that OS^v is able to efficiently run a variety of existing applications. We demonstrate its sub-second boot time, small OS image and how it makes more memory available to the application. For unmodified network-intensive applications, we demonstrate up to 25% increase in throughput and 47% decrease in latency. By using non-POSIX network APIs, we can further improve performance and demonstrate a 290% increase in Memcached throughput.

1 Introduction

Cloud computing (Infrastructure-as-a-Service, or IaaS) was born out of the realization that virtualization makes it easy and safe for different organizations to share one pool of physical machines. At any time, each organization can rent only as many virtual machines as it currently needs to run its application.

Today, virtual machines on the cloud typically run the same traditional operating systems that were used on physical machines, e.g., Linux, Windows, and *BSD. But as the IaaS cloud becomes ubiquitous, this choice is starting to make less sense: The features that made these operating systems desirable on physical machines, such as familiar single-machine administration interfaces and support for a large selection of hardware, are losing their relevance. At the same time, different features are becoming important: The VM's operating system needs to be fast, small, and easy to administer at large scale.

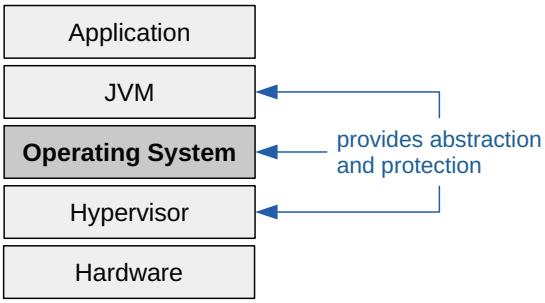

Figure 1: Software layers in a typical cloud VM.

Moreover, fundamental features of traditional operating systems are becoming overhead, as they are now duplicated by other layers of the cloud stack (illustrated in Figure 1).

For example, an important role of traditional operating systems is to isolate different processes from one another, and all of them from the kernel. This isolation comes at a cost, in performance of system calls and context switches, and in complexity of the OS. This was necessary when different users and applications ran on the same OS, but on the cloud, the hypervisor provides isolation between different VMs so mutually-untrusting applications do not need to run on the same VM. Indeed, the scale-out nature of cloud applications already resulted in a trend of focused single-application VMs.

A second example of duplication is hardware abstraction: An OS normally provides an abstraction layer through which the application accesses the hardware. But on the cloud, this "hardware" is itself a virtualized abstraction created by the hypervisor. Again, this duplication comes at a performance cost.

This paper explores the question of what an operating system would look like if we designed it today with the sole purpose of running on virtual machines on the cloud, and not on physical machines.

We present OSv, a new OS we designed specifically for cloud VMs. The main goals of OSv are as follows:

- Run existing cloud applications (Linux executables).
- Run these applications faster than Linux does.
- Make the image small enough, and the boot quick enough, that starting a new VM becomes a viable alternative to reconfiguring a running one.
- Explore new APIs for new applications written for OSv, that provide even better performance.
- Explore using such new APIs in common runtime environments, such as the Java Virtual Machine (JVM). This will boost the performance of unmodified Java applications running on OSv.
- Be a platform for continued research on VM operating systems. OSv is actively developed as open source, it is written in a modern language (C++11), its codebase is relatively small, and our community encourages experimentation and innovation.

OSv supports different hypervisors and processors, with only minimal amount of architecture-specific code. For 64-bit x86 processors, it currently runs on the KVM, Xen, VMware and VirtualBox hypervisors, and also on the Amazon EC2 and Google GCE clouds (which use a variant of Xen and KVM, respectively). Preliminary support for 64-bit ARM processors is also available.

In Section 2, we present the design and implementation of OSv. We will show that OSv runs only on a hypervisor, and is well-tuned for this (e.g., by avoiding spinlocks). OSv runs a single application, with the kernel and multiple threads all sharing a single address space. This makes system calls as efficient as function calls, and context switches quicker. OSv supports SMP VMs, and has a redesigned network stack (*network channels*) to lower socket API overheads. OSv includes other facilities one expects in an operating system, such as standard libraries, memory management and a thread scheduler, and we will briefly survey those. OSv's scheduler incorporates several new ideas including lock-free algorithms and floating-point based fair accounting of run-time.

In Section 3, we begin to explore what kind of new APIs a single-application OS like OSv might have beyond the traditional POSIX APIs to further improve performance. We suggest two techniques to improve JVM memory utilization and garbage-collection performance, which boost performance of all JVM languages (Java, Scala, Jruby, etc.) on OSv. We then demonstrate that a zero-copy, lock-free API for packet processing can result in a 4x increase of Memcached throughput.

In Section 4, we evaluate our implementation, and compare OSv to Linux on several micro- and macrobenchmarks. We show minor speedups over Linux in computation- and memory-intensive workloads such as the SPECjvm2008 benchmark, and up to 25% increase in throughput and 47% reduction in latency in network-dominated workloads such as Netperf and Memcached.

2 Design and Implementation of OSv

OSv follows the *library OS* design, an OS construct pioneered by *exokernels* in the 1990s [5]. In OSv's case, the hypervisor takes on the role of the exokernel, and VMs the role of the applications: Each VM is a single application with its own copy of the library OS (OSv). Library OS design attempts to address performance and functionality limitations in applications that are caused by traditional OS abstractions. It moves resource management to the application level, exports hardware directly to the application via safe APIs, and reduces abstraction and protection layers.

OSv runs a single application in the VM. If several mutually-untrusting applications are to be run, they can be run in separate VMs. Our assumption of a single application per VM simplifies OSv, but more importantly, eliminates the redundant and costly isolation inside a guest, leaving the hypervisor to do isolation. Consequently, OSv uses a *single address space* — all threads and the kernel use the same page tables, reducing the cost of context switches between applications threads or between an application thread and the kernel.

The OSv kernel includes an ELF dynamic linker which runs the desired application. This linker accepts standard ELF dynamically-linked code compiled for Linux. When this code calls functions from the Linux ABI (i.e., functions provided on Linux by the glibc library), these calls are resolved by the dynamic linker to functions implemented by the OSv kernel. Even functions which are considered "system calls" on Linux, e.g., `read()`, in OSv are ordinary function calls and do not incur special call overheads, nor do they incur the cost of user-to-kernel parameter copying which is unnecessary in our single-application OS.

Aiming at compatibility with a wide range of existing applications, OSv emulates a big part of the Linux programming interface. Some functions like `fork()` and `exec()` are not provided, since they don't have any meaning in the one-application model employed by OSv.

The core of OSv is new code, written in C++11. This includes OSv's loader and dynamic linker, memory management, thread scheduler and synchronization mecha-

nisms such as mutex and RCU, virtual-hardware drivers, and more. We will discuss below some of these mechanisms in more detail.

Operating systems designed for physical machines usually devote much of their code to supporting diverse hardware. The situation is much easier for an operating system designed for VMs, such as OSv, because hypervisors export a simplified and more stable hardware view. OSv has drivers for a small set of traditional PC devices commonly emulated by hypervisors, such as a keyboard, VGA, serial port, SATA, IDE and HPET. Additionally, it supports several paravirtual drivers for improved performance: A paravirtual clock is supported on KVM and Xen, a paravirtual NIC using virtio [25] and VMXNET3 [29], and a paravirtual block device (disk) using virtio and pvscsi.

For its filesystem support, OSv follows a traditional UNIX-like VFS (virtual filesystem) design [12] and adopts ZFS as its major filesystem. ZFS is a modern filesystem emphasizing data integrity and advanced features such as snapshots and volume management. It employs a modified version of the Adaptative Replacement Cache [18] for page cache management and consequently it can achieve a good balance between recency and frequency hits.

Other filesystems are also present in OSv. There is one in-memory filesystem for specialized applications that may want to boot without disk (ramfs), and a very simple device filesystem for device views (devfs). For compatibility with Linux applications, a simplified procfs is also supported.

Some components of OSv were not designed from scratch, but rather imported from other open-source projects. We took the C library headers and some functions (such as stdio and math functions) from the *musl libc* project, the VFS layer from *Prex* project, the ZFS filesystem from *FreeBSD*, and and the ACPI drivers from the *ACPICA* project. All of these are areas in which OSv's core value is not expected to be readily apparent so it would make less sense for these to be written from scratch, and we were able to save significant time by reusing existing implementations.

OSv's network stack was also initially imported from FreeBSD, because it was easier to start with an implementation known to be correct, and later optimize it. As we explain in Section 2.3, after the initial import we rewrote the network stack extensively to use a more efficient *network channels*-based design.

It is beyond the scope of this article to cover every detail of OSv's implementation. Therefore, the remainder of this section will explore a number of particularly interesting or unique features of OSv's implementation, including: 1. memory management in OSv, 2. how and why OSv completely avoids spinlocks, 3. network channels, a non-traditional design for the networking stack, and 4. the OSv thread scheduler, which incorporates several new ideas including lock-free algorithms and floating-point based fair accounting of run-time.

2.1 Memory Management

In theory, a library OS could dictate a flat physical memory mapping. However, OSv uses virtual memory like general purpose OSs. There are two main reasons for this. First, the x86_64 architecture mandates virtual memory usage for long mode operation. Second, modern applications following traditional POSIX-like APIs tend to map and unmap memory and use page protection themselves.

OSv supports demand paging and memory mapping via the `mmap` API. This is important, for example, for a class of JVM-based applications that bypass the JVM and use `mmap` directly through JNI. Such applications include Apache Cassandra which is a popular NoSQL database running on the JVM.

For large enough mappings, OSv will fill the mapping with huge pages (2MB in size for the x86_64 architecture). The use of larger page sizes improve performance of applications by reducing the number of TLB misses. [24].

Since mappings can be partially unmapped, it is possible that one of these pages needs to be broken into smaller pages. By employing a mechanism similar to Linux's Transparent Huge Pages, OSv handles this case transparently.

As an OS that aims to support a single application, page eviction is not supported. Additional specialized memory management constructs are described in Section 3.

2.2 No Spinlocks

One of the primitives used by contemporary OSs on SMP machines is the spin-lock [2]. On a *single*-processor system, it is easy to protect a data structure from concurrent access by several contexts by disabling interrupts or context switches while performing non-atomic modifications. That is not enough on *multi*-processor systems, where code running on multiple CPUs may touch the data concurrently. Virtually all modern SMP OSs today use spin-locks: One CPU acquires the lock with an atomic test-and-set operation, and the others execute a busy-loop until they can acquire the lock themselves. SMP OSs use this spin-lock primitive to implement higher-level locking facilities such as *sleeping mutexes*, and also use spin-locks directly in situations where sleeping is forbidden, such as in the scheduler itself and in interrupt-handling context.

Spin-locks are well-suited to a wide range of SMP physical hardware. However when we consider *virtual* machines, spin-locks suffer from a significant drawback known as the "lock-holder preemption" problem [28]: while physical CPUs are always running if the OS wants them to, virtual CPUs may "pause" at unknown times for unknown durations. This can happen during exits to the hypervisor or because the hypervisor decides to run other guests or even hypervisor processes on this CPU.

If a virtual CPU is paused while holding a spin-lock, other CPUs that want the same lock spin needlessly, wasting CPU time. When a mutex is implemented using a spin-lock, this means that a thread waiting on a lock can find itself spinning and wasting CPU time, instead of immediately going to sleep and letting another thread run. The consequence of the lock-holder preemption problem is lower performance — Friebel et al. have shown that multitasking two guests on the same CPU results in performance drops from 7% up to 99% in extreme cases [7].

Several approaches have been proposed to mitigate the lock-holder preemption problem [7], usually requiring changes to the hypervisor or some form of cooperation between the hypervisor and the guest. However, in a kernel designed especially to run in a virtual machine, a better solution is to avoid the problem completely. OS^v does not use spin-locks *at all*, without giving up on lock-based algorithms in the kernel or restricting it to single-processor environments.

One way to eliminate spin-locks is to use lock-free algorithms [19]. These algorithms make clever use of various atomic instructions provided by the SMP machine (e.g., *compare-exchange*, *fetch-and-add*) to ensure that a data structure remains in consistent state despite concurrent modifications. We can also avoid locks by using other techniques such as *Read-Copy-Update* (RCU) [17]. But lock-free algorithms are very hard to develop, and it is difficult to completely avoid locking in the kernel [16], especially considering that we wanted to re-use existing kernel components such as ZFS and the BSD network stack. Therefore, our approach is as follows:

1. Ensure that most work in the kernel, including interrupt handling, is done in threads. These can use lock-based algorithms: They use a mutex (which can put a thread to sleep), not a spin-lock.

2. Implement the mutex itself without using a spin-lock, i.e., it is a lock-free algorithm.

3. The scheduler itself cannot be run in a thread, so to protect its data structures without spin-locks, we use per-cpu run queues and lock-free algorithms.

OS^v executes almost everything in ordinary threads. Interrupt handlers usually do nothing but wake up a thread which will service the interrupting device. Kernel code runs in threads just like application code, and can sleep or be preempted just the same. OS^v's emphasis on cheap thread context switches ensures that the performance of this design does not suffer.

Our mutex implementation is based on a lock-free design by Gidenstam & Papatriantafilou [8], which protects the mutex's internal data structures with atomic operations in a lock-free fashion. With our lock-free mutex, a paused CPU cannot cause other CPUs to start spinning. As a result, kernel and application code which uses this mutex are free from the lock-holder preemption problem.

Finally, the scheduler itself uses per-CPU run queues, so that most scheduling decisions are local to the CPU and need no locking. It uses lock-free algorithms when scheduling cooperation is needed across CPUs, such as waking a thread that belongs to a different CPU. OS^v's scheduler is described in more detail in Section 2.4.

2.3 Network Channels

An operating system designed for the cloud must, almost by definition, provide a high quality TCP/IP stack. OS^v does this by applying Van Jacobson's *net channel* ideas [10] to its networking stack.

We begin by observing that a typical network stack is traversed in two different directions:

- Top-down: the `send()` and `recv()` system calls start at the socket layer, convert user buffers to TCP packets, attach IP headers to those TCP packets, and finally egress via the network card driver,

- Bottom-up: incoming packets are received by the network card driver, parsed by the IP layer, forwarded to the TCP layer, and are then appended to socket buffers; blocked `send()`, `recv()`, and `poll()` system calls are then woken as necessary.

As illustrated in Figure 2a, both the interrupt contexts (hard- and soft- interrupt) and the application thread context perform processing on all layers of the network stack. The key issue is that code from both contexts accesses shared data structures, causing lock and cache-line contention on heavily used connections.

In order to resolve this contention, under OS^v almost all packet processing is performed in an application thread. Upon packet receipt, a simple classifier associates it with a *channel*, which is a single producer/single consumer queue for transferring packets to the application thread. Each channel corresponds to a single flow, such as a TCP connection or a UDP path from an interface to a socket.

As can be seen in Figure 2b, access to shared data structures from multiple threads is completely eliminated (save for the channel itself).

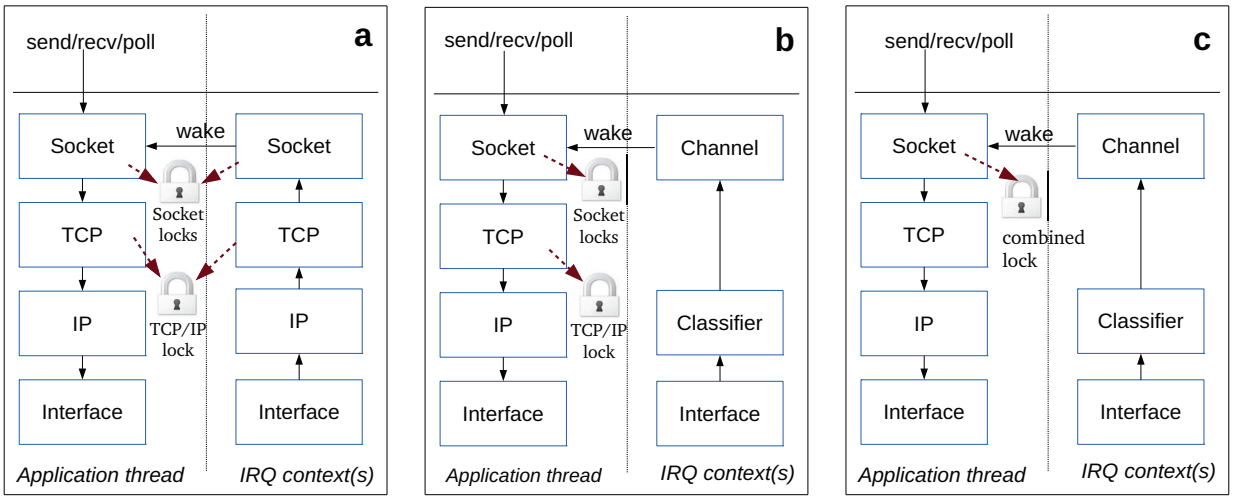

Figure 2: Control flow and locking in (left to right): (a) a traditional networking stack, (b) OSv's networking stack prior to lock merging, and (c) OSv's complete networking stack

In addition, since there is now just one thread accessing the data, locking can be considerably simplified, reducing both run-time and maintenance overhead.

Switching to a net channel approach allows a significant reduction in the number of locks required, leading to the situation in Figure 2c:

- The socket receive buffer lock has been merged with the socket send buffer lock; since both buffers are now populated by the same thread (running either the `send()` or `recv()` system calls), splitting that lock is unnecessary,
- The interleave prevention lock (used to prevent concurrent writes from interleaving) has been eliminated and replaced by a wait queue using the socket lock for synchronization, and
- The TCP layer lock has been merged with the socket layer lock; since TCP processing now always happens within the context of a socket call, it is already protected by that lock.

We expect further simplifications and improvements to the stack as it matures.

2.4 The Thread Scheduler

The guiding principles of OSv's thread scheduler are that it should be *lock-free*, *preemptive*, *tick-less*, *fair*, *scalable* and *efficient*.

Lock-free As explained in Section 2.2, OSv's scheduler should not use spin-locks and it obviously cannot use a sleeping mutex.

The scheduler keeps a separate *run queue* on each CPU, listing the *runnable* threads on the CPU. Sleeping threads are not listed on any run queue. The scheduler runs on a CPU when the running thread asks for a reschedule, or when a timer expiration forces preemption. At that point, the scheduler chooses the most appropriate thread to run next from the threads on this CPU's run-queue, according to its fairness criteria. Because each CPU has its own separate run-queue, this part of the scheduler needs no locking.

The separate run queues can obviously lead to a situation where one CPU's queue has more runnable threads than another CPU's, hurting the scheduler's overall fairness. We solve this by running a *load balancer* thread on each CPU. This thread wakes up once in a while (10 times a second), and checks if some other CPU's run queue is shorter than this CPU's. If it is, it picks one thread from this CPU's run queue, and wakes it on the remote CPU.

Waking a thread on a remote CPU requires a more elaborate lock-free algorithm: For each of the N CPUs, we keep N lock-free queues of *incoming wakeups*, for a total of N^2 queues. We also keep a bitmask of nonempty queues for each CPU. When CPU s wants to wake a thread on CPU d, it adds this thread to the queue (s,d), atomically turns on bit s in CPU d's bitmask and sends an inter-processor interrupt (IPI) to CPU d. The interrupt leads CPU d to perform a reschedule, which begins by looking for incoming wakeups. The bitmask tells the scheduler which of the incoming queues it needs to inspect.

Preemptive OSv fully supports preemptive multitasking: While threads can voluntarily cause a reschedule (by waiting, yielding, or waking up another thread), one can also happen at any time, preempted by an interrupt such as a timer or the wakeup IPI mentioned above. All threads are preemptable and as with the rest of the system, there is no difference between application and kernel threads. A thread can temporarily avoid being preempted by incrementing a per-thread preempt-disable counter. This feature can be useful in a number of cases including, for example, maintaining per-CPU variables and RCU [17] locks. An interrupt while the running thread has preemption disabled will not cause a reschedule, but when the thread finally re-enables preemption, a reschedule will take place.

Tick-less Most classic kernels, and even many modern kernels, employ a periodic timer interrupt, also known as a *tick*. The tick causes a reschedule to happen periodically, for example, 100 times each second. Such kernels often account the amount of time that each thread has run in whole ticks, and use these counts to decide which thread to schedule at each tick.

Ticks are convenient, but also have various disadvantages. Most importantly, excessive timer interrupts waste CPU time. This is especially true on on virtual machines where interrupts are significantly slower than on physical machines, as they involve exits to the hypervisor.

Because of the disadvantages of ticks, OSv implements a tickless design. Using a high resolution clock, the scheduler accounts to each thread the exact time it consumed, instead of approximating it with ticks. Some timer interrupts are still used: Whenever the fair scheduling algorithm decides to run one thread, it also calculates when it will want to switch to the next thread, and sets a timer for that period. The scheduler employs hysteresis to avoid switching too frequently between two busy threads. With the default hysteresis setting of 2ms, two busy threads with equal priority will alternate 4ms time slices, and the scheduler will never cause more than 500 timer interrupts each second. This number will be much lower when there aren't several threads constantly competing for CPU.

Fair On each reschedule, the scheduler must decide which of the CPU's runnable threads should run next, and for how long. A fair scheduler should account for the amount of *run time* that each thread got, and strive to either equalize it or achieve a desired ratio if the threads have different priorities. However, using the total runtime of the threads will quickly lead to imbalances. For instance, if a thread was out of the CPU for 10 seconds and becomes runnable, it will monopolize the CPU for 10 whole seconds as the scheduler seeks to achieve fairness. Instead, we want to equalize the amount of run-time that runnable threads have gotten in recent history, and forget about the distant past.

OSv's scheduler calculates the exponentially-decaying moving average of each thread's recent run time. The scheduler will choose to run next the runnable thread with the lowest moving-average runtime, and calculate exactly how much time this thread should be allowed to run before its runtime surpasses that of the runner-up thread.

Our moving-average runtime is a floating-point number. It is interesting to mention that while some kernels forbid floating-point use inside the kernel, OSv fully allows it. As a matter of fact, it has no choice but to allow floating point in the kernel due to the lack of a clear boundary between the kernel and the application.

The biggest stumbling block to implementing moving-average runtime as described above is its scalability: It would be impractical to update the moving-average runtimes of *all* threads on each scheduler invocation.

But we can show that this is not actually necessary; we can achieve the same goal with just updating the runtime of the single running thread. It is beyond the scope of this article to derive the formulas used in OSv's scheduler to maintain the moving-average runtime, or to calculate how much time we should allow a thread to run until its moving-average runtime overtakes that of the runner-up thread.

Scalable OSv's scheduler has $O(\lg N)$ complexity in the number of runnable threads on each CPU: The run queue is kept sorted by moving-average runtime, and as explained, each reschedule updates the runtime of just one thread. The scheduler is totally unaware of threads which are not runnable (e.g., waiting for a timer or a mutex), so there is no performance cost in having many utility threads lying around and rarely running. OSv indeed has many of these utility threads, such as the load-balancer and interrupt-handling threads.

Efficient Beyond the scheduler's scalability, OSv employs additional techniques to make the scheduler and context switches more efficient.

Some of these techniques include:

- OSv's single address space means that we do not need to switch page tables or flush the TLB on context switches. This makes context switches significantly cheaper than those on traditional multi-process operating systems.
- Saving the floating-point unit (FPU) registers on every context switch is also costly. We make use of the fact that most reschedules are voluntary, caused

by the running thread calling a function such as `mutex_wait()` or `wake()`. The x86_64 ABI guarantees that the FPU registers are caller-saved. So for voluntary context switches, we can skip saving the FPU state.

As explained above, waking a sleeping thread on a different CPU requires an IPI. These are expensive, and even more so on virtual machines, where both sending and receiving interrupts cause exits to the hypervisor. As an optimization, idle CPUs spend some time before halting in polling state, where they ask not to be sent these IPIs, and instead poll the wakeup bitmask. This optimization can almost eliminate the expensive IPIs in the case where two threads on two CPUs wait for one another in lockstep.

3 Beyond the Linux APIs

In this section, we explore what kind of new APIs a single-application OS like OSv might have beyond the standard Linux APIs, and discuss several such extensions which we have already implemented as well as their benefits.

The biggest obstacle to introducing new APIs is the need to modify existing applications or write new applications. One good way around this problem is to focus on efficiently running a *runtime environment*, such as the Java Virtual Machine (JVM), on OSv. If we optimize the JVM itself, any application run inside this JVM will benefit from this optimization.

As explained in the previous section, OSv can run unmodified Linux programs, which use the Linux APIs — a superset of the POSIX APIs. We have lowered the overhead of these APIs, as described in the previous section and quantified in the next section. One of the assumptions we have made is that OSv runs a single application, in a single address space. This allowed us to run "system calls" as ordinary functions, reducing their overhead.

However, in this section we show that there remain significant overheads and limitations inherent in the Linux APIs, which were designed with a multi-process multi-user operating system in mind. We propose to reduce these remaining overheads by designing new APIs specifically for applications running on a single-process OS like OSv.

The socket API, in particular, is rife with such overheads. For example, a socket read or write necessarily copies the data, because on Linux the kernel cannot share packet buffers with user space. But on a single-address-space OS, a new zero-copy API can be devised where the kernel and user space share the buffers. For packet-processing applications, we can adopt a netmap-like API [23]. The OSv kernel may even expose the host's virtio rings to the application (which is safe when we have a single application), completely eliminating one layer of abstraction. In Section 4 we demonstrate a Memcached implementation which uses a non-POSIX packet processing API to achieve a 4-fold increase of throughput compared to the traditional Memcached using the POSIX socket APIs.

Another new API benefiting from the single-application nature of OSv is one giving the application direct access to the page table. Java's GC performance, in particular, could benefit: The Hotspot JVM uses a data structure called a *card table* [22] to track write accesses to references to objects. To update this card table to mark memory containing that reference as dirty, the code generated by the JVM has to be followed by a "write barrier". This additional code causes both extra instructions and cache line bounces. However, the MMU already tracks write access to memory. By giving the JVM access to the MMU, we can track reference modifications without a separate card table or write barriers. A similar strategy is already employed by Azul C4 [27], but it requires heavy modifications to the Linux memory management system.

In the rest of this section, we present two new non-Linux features which we implemented in OSv. The first feature is a *shrinker* API, which allows the application and the kernel to share the entire available memory. The second feature, the *JVM balloon*, applies the shrinker idea to an unmodified Java Virtual Machine, so that instead of manually chosing a heap size for the JVM, the heap is automatically resized to fill all memory which the kernel does not need.

3.1 Shrinker

The shrinker API allows the application or an OS component to register callback functions that OSv will call when the system is low on memory. The callback function is then responsible for freeing some of that application or component's memory. Under most operating systems, applications or components that maintain a dynamic cache, such as a Memcached cache or VFS page cache, must statically limit its size to a pre-defined amount of memory or number of cache entries. This imposes sometimes contradicting challenges: not to consume more memory than available in the system and not to strangle other parts of the system, while still taking advantage of the available memory. This gets even more challenging when there are a few heavy memory consumers in the system that work in a bursty manner wherein the memory needs to "flow" from one application or component to another depending on demand. The shrinker API provides an adaptable solution by allowing applications and components to handle memory pressure

as it arises, instead of requiring admininstrators to tune in advance.

We have demonstrated the usefulness of the shrinker API in two cases — Memcached [6], and the JVM. Ordinarily, Memcached requires the in-memory cache size to be specified (with the "-m" option) and the JVM requires the maximum heap size to be specified (the "-Xmx" option). Setting these sizes manually usually results in wasted VM memory, as the user decreases the cache or heap size to leave "enough" memory to the OS. Our Memcached re-implementation described in Section 4 uses the shrinker API and does not need the "-m" option: it uses for its cache all the memory which OS^v doesn't need. We can similarly modify the JVM to use the shrinker to automatically size its heap, and even achieve the same on an unmodified JVM, as we will explain now.

3.2 JVM Balloon

The *JVM balloon* is a mechanism we developed to automatically determine the JVM heap size made available to the application. Ballooning is a widely used mechanism in hypervisors [30, 31] and the JVM balloon draws from the same core idea: providing efficient dynamic memory placement and reducing the need to do complex planning in advance. OS^v's JVM balloon is designed to work with an unmodified JVM. As a guest-side solution, it will also work on all supported hypervisors.

It is possible to modify the JVM code to simplify this process. But the decision to run it from the OS side allows for enhanced flexibility, since it avoids the need to modify the various extant versions and vendor-implementations of the JVM.

The JVM allocates most of its memory from its heap. This area can grow from its minimum size but is bounded by a maximum size, both of which can be specified by initialization parameters. The size of the JVM heap directly influences performance for applications since having more memory available reduces occurrences of GC cycles.

However, a heap size that is too big can also hurt the application since the OS will be left without memory to conduct its tasks — like buffering a large file — when it needs to. Although any modern OS is capable of paging through the virtual-memory system, the OS usually lacks information during this process to make the best placement decision. A normal OS will see all heap areas as pages whose contents cannot be semantically interpreted. Consequently, it is forced to evict such pages to disk, which generates considerable disk activity and suboptimal cache growth. At this point an OS that is blind to the semantic content of the pages will usually avoid evicting too much since it cannot guarantee that those pages will not be used in the future. This results in less memory being devoted to the page cache, where it would potentially bring the most benefit. We quantify this effect in Section 4, and show that OS^v's JVM balloon allows pages to be discarded without any disk activity.

OS^v's approach is to allocate almost all available memory to the JVM when it is started [1], therefore setting that memory as the *de facto* JVM maximum heap. The OS allocations can proceed normally until pressure criteria are met.

Upon pressure, OS^v will use JNI [13] to create an object in the JVM heap with a size big enough to alleviate that pressure and acquire a reference to it. The object chosen is a `ByteArray`, since these are laid down contiguously in memory and it is possible to acquire a pointer to their address from JNI.

This object is referenced from the JNI, so a GC will not free it and at this point the heap size is effectively reduced by the size of the object, forcing the JVM to count on a smaller heap for future allocations. Because the balloon object still holds the actual pages as backing storage, the last step of the ballooning process is to give the pages back to the OS by unmapping that area. The JVM cannot guarantee or force any kind of alignment for the object, which means that in this process some memory will be wasted: it will neither be used the Java application nor given back to the OS. To mitigate this we use reasonably large minimum balloon sizes (128MB).

Balloon movement

The reference to the object, held by OS^v, guarantees that the object will not be disposed by the JVM or taken into account when making collection decisions. However, it does not guarantee that the object is never touched again. When the JVM undergoes a GC cycle, it moves the old objects to new locations to open up space for new objects to come. At this point, OS^v encounters a page fault.

OS^v assumes that nothing in the JVM directly uses that object, and therefore is able to make the following assumptions about page faults that hit the balloon object:

- all reads from it are part of a copy to a new location,
- the source and destination addresses correspond to the same offset within the object,
- whenever that region is written to, it no longer holds the balloon.

With that in mind, OS^v's page fault handler can decode the copy instruction — usually a `rep mov` in x86 — and find its destination operand. It then recreates the balloon in the destination location and updates all register values

[1] 90% in the current implementation

to make the copier believe the copy was successfully conducted. OSv's balloon mechanism is expected to work with any JVM or collector in which these assumptions hold.

The old location is kept unmapped until it is written to. This has both the goal of allowing the remap to be lazy, and to correctly support GCs that may speculatively copy the object to more than one location. Such is the case, for instance, for OpenJDK's Parallel Scavenge Garbage Collector.

4 Evaluation

We conducted some experiments to measure the performance of OSv as a guest operating system, and demonstrate improvement over a traditional OS: Linux. In all runs below, for "Linux" we used a default installation of Fedora 20 with the `iptables` firewall rules cleared. We look at both micro-benchmarks measuring the performance of one particular feature, and macro-benchmarks measuring the overall performance of an application.

The host used in the benchmarks was a 4-CPU 3.4GHz Intel® Core™ i7-4770 CPU, 16GB of RAM, with an SSD disk. The host was running Fedora 20 Linux and the KVM hypervisor.

Macro Benchmarks

Memcached is a high-performance in-memory key-value storage server [6]. It is used by many high-profile Web sites to cache results of database queries and prepared page sections, to significantly boost these sites' performance. We used the *Memaslap* benchmark to load the server and measure its performance. Memaslap runs on a remote machine (connected to the tested host with a direct 40 GbE cable), sends random requests (concurrency 120), 90% `get` and 10% `set`, to the server and measures the request completion rate. In this test, we measured a single-vCPU guest running Memcached with one service thread. Memcached supports both UDP and TCP protocols — we tested the UDP protocol which is considered to have lower latency and overhead [20]. We set the combination of Memcached's cache size (5 GB) and memaslap test length (30 seconds) to ensure that the cache does not fill up during the test.

Table 1 presents the results of the *memaslap* benchmark, comparing the same unmodified Memcached program running on OSv and Linux guests. We can see that Memcached running on OSv achieves 22% higher throughput than when running on Linux.

One of the stated goals of OSv was that an OSv guest boots quickly, and has a small image size. Indeed, we measured the time to boot OSv and Memcached, until Memcached starts serving requests, to be just 0.6 seconds. The guest image size was just 11MB. We believe that both numbers can be optimized further, e.g., by using ramfs instead of ZFS (Memcached does not need persistent storage).

Guest OS	Transactions / sec	Score
Linux	104394	1
OSv	127275	1.22

Table 1: Memcached and Memaslap benchmark

In Section 3 we proposed to further improve performance by implementing in OSv new networking APIs with lower overheads than the POSIX socket APIs. To test this direction, we re-implemented part of the Memcached protocol (the parts that the memaslap benchmark uses). We used a packet-filtering API to grab incoming UDP frames, process them, and send responses in-place from the packet-filter callback. As before, we ran this application code in a single-vCPU guest running OSv and measured it with memaslap. The result was 406750 transactions/sec — 3.9 times the throughput of the baseline Memcached server on Linux.

SPECjvm2008 is a Java benchmark suite containing a variety of real-life applications and benchmarks. It focuses on the performance of the JVM executing a single application, and reflects the performance of CPU- and memory-intensive workloads, having low dependence on file I/O and including no network I/O across machines.

SPECjvm2008 is not only a performance benchmark, it is also a good correctness test for OSv. The benchmarks in the suite use numerous OS features, and each benchmark validates the correctness of its computation.

Table 2 shows the scores for both OSv and Linux for the SPECjvm2008 benchmarks. For both guest OSs, the guest is given 2GB of memory and two vCPUs, and the benchmark is configured to use two threads. The JVM's heap size is set to 1400MB.

Benchmark	OSv	Linux	Benchmark	OSv	Linux
Weighted average	**1.046**	**1.041**	sor.large	27.3	27.1
compiler.compiler	377	393	sparse.large	27.7	27.2
compiler.sunflow	140	149	fft.small	138	114
compress	111	109	lu.small	216	249
crypto.aes	57	56	sor.small	122	121
crypto.rsa	289	279	sparse.small	159	163
crypto.signverify	280	275	monte-carlo	159	150
derby	176	181	serial	107	107
mpegaudio	100	100	sunflow	56.6	55.4
fft.large	35.5	32.8	xml.transform	251	247
lu.large	12.2	12.2	xml.validation	480	485

Table 2: SPECjvm2008 — higher is better

We did not expect a big improvement, considering that SPECjvm2008 is computation-dominated with relatively little use of OS services. Indeed, on average, the SPECjvm2008 benchmarks did only 0.5% better on OSv

than on Linux. This is a small but statistically-significant improvement (the standard deviation of the weighted average was only 0.2%). OSv did slightly worse than Linux on some benchmarks (notably those relying on the filesystem) and slightly better on others. We believe that with further optimizations to OSv we can continue to improve its score, especially on the lagging benchmarks, but the difference will always remain small in these computation-dominated benchmarks.

Micro Benchmarks

Network performance: We measured the network stack's performance using the Netperf benchmark [11] running on the host. Tables 3 and 4 shows the results for TCP and UDP tests respectively. We can see that OSv consistently outperforms Linux in the tests. RR (request/response) is significantly better for both TCP and UDP, translating to 37%-47% reduction in latency. TCP STREAM (single-stream throughput) is 24%-25% higher for OSv.

Test	STREAM (Mbps)	RR (Tps)
Linux UP	44546 ± 941	45976 ± 299
Linux SMP	40149 ± 1044	45092 ± 1101
OSv UP	55466 ± 553	74862 ± 405
OSv SMP	49611 ± 1442	72461 ± 572

Table 3: Netperf TCP benchmarks: higher is better

Test	RR (Tps)
Linux UP	44173 ± 345
Linux SMP	47170 ± 2160
OSv UP	82701 ± 799
OSv SMP	74367 ± 1246

Table 4: Netperf UDP benchmarks: higher is better

JVM balloon: To isolate the effects of the JVM balloon technique described in Section 3.2, we wrote a simple microbenchmark in Java to be run on both Linux and OSv. It consists of the following steps:

1. Allocate 3.5 GB of memory in 2MB increments and store them in a list,
2. Remove from the list and write each 2MB buffer to a file sequentially until all buffers are exhausted,
3. Finally read that file back to memory.

In both guest OSs, the application ran alone in a VM with 4GB of RAM. For OSv, the JVM heap size was automatically calculated by the balloon mechanism to 3.6 GB. For Linux, the same value was manually set.

As shown in Table 5, OSv fared better in this test than Linux by around 35%. After the first round of allocations the guest memory is almost depleted. As Linux needs more memory to back the file it has no option but to evict JVM heap pages. That generates considerable disk activity, that not only is detrimental per se, but will in this particular moment compete with the application disk writes.

We observe that not only is the execution slower on Linux, it also has a much higher standard deviation. This is consistent with our expectation. Aside from deviations arising from the I/O operations themselves, the Linux VM lacks information to make the right decision about which pages is best to evict.

Guest OS	Total (sec)	File Write (sec)	File Read (sec)
Linux	40 ± 6	27 ± 6	10.5 ± 0.2
OSv	26 ± 1	16 ± 1	7.4 ± 0.2

Table 5: JVM balloon micro-benchmark: lower is better

OSv can be more agressive when discarding pages because it doesn't have to evict pages to make room for the page cache, while Linux will be a lot more conservative in order to avoid swap I/O. That also speeds up step 3 ("File Read"), as can be seen in Table 5. In the absence of eviction patterns, both Linux and OSv achieve consistent results with a low deviation. However, Linux reaches this phase with a smaller page cache to avoid generating excessive disk activity. OSv does not need to make such compromise, leading to a 30% performance improvement in that phase alone.

Context switches: We wrote a context-switch microbenchmark to quantify the claims made earlier that thread switching is significantly cheaper on OSv than it is on Linux. The benchmark has two threads, which alternate waking each other with a pthreads condition variable. We then measure the average amount of time that each such wake iteration took.

The benchmark is further subdivided into two cases: In the "colocated" case, the two alternating threads are colocated on the same processor, simulating the classic uniprocessor context switch. In the "apart" case, the two threads are pinned to different processors.

Guest OS	Colocated	Apart
Linux	905 ns	13148 ns
OSv	328 ns	1402 ns

Table 6: Context switch benchmark

The results are presented in Table 6. It shows that thread switching is indeed much faster in OSv than in Linux — between 3 and 10 times faster. The "apart" case is especially helped in OSv by the last optimization described in 2.4, of idle-time polling.

5 Related Work

Containers [26, 3] use a completely different approach to eliminate the feature duplication of the hypervisor and guest OS. They abandon the idea of a hypervisor, and instead provide *OS-level virtualization* — modifying the host OS to support isolated execution environments for applications while sharing the same kernel. This approach improves resource sharing between guests and lowers per-guest overhead. Nevertheless, the majority of IaaS clouds today use hypervisors. These offer tenants better-understood isolation and security guarantees, and the freedom to choose their own kernel.

Picoprocesses [4] are another contender to replace the hypervisor. While a containers' host exposes to its guests the entire host kernel's ABI, picoprocesses offer only a bare-minimum API providing basic features like allocating memory, creating a thread and sending a packet. On top of this minimal API, a library OS is used to allow running executables written for Linux [9] or Windows [21]. These library OSs are similar to OS^v in that they take a minimal host/guest interface and use it to implement a full traditional-OS ABI for a single application, but the implementation is completely different. For example, the picoprocess POSIX layer uses the host's threads, while OS^v needs to implement threads and schedule these threads on its own.

If we return our attention to hypervisors, one known approach to reducing the overheads of the guest OS is to take an existing operating system, such as a Linux distribution, and trim it down as much as possible. Two examples of this approach are CoreOS and Tiny Core Linux. OS^v differs from these OSs in that it is a newly designed OS, not a derivative of Linux. This allowed OS^v to make different design decisions than Linux made, e.g., our choice not to use spinlocks, or to have a single address space despite having an MMU.

While OS^v can run applications written in almost any language (both compiled and high-level), some VM OSs focus on running only a single high-level language. For example, *Erlang on Xen* runs an Erlang VM directly on the Xen hypervisor. *Mirage OS* [14] is a library OS written in OCaml that runs on the Xen hypervisor. It takes the idea of a library OS to the extreme where an application links against separate OS service libraries and unused services are eliminated from the final image by the compiler. For example, a DNS server VM image can be as small as 200 KB.

Libra [1] is a library OS for running IBM's J9 JVM in a VM. Libra makes the case that as JVM already has sandboxing, a memory model, and a threading model, a general purpose OS is redundant. However, Libra does not replace the whole OS but instead relies on Linux running in a separate hypervisor partition to provide networking and filesystem.

ClickOS [15] is an optimized operating system for VMs specializing in network processing applications such as routing, and achieves impressive raw packet-per-second figures. However, unlike OS^v which runs on multiple hypervisors, ClickOS can only run on Xen, and requires extensive modifications to Xen itself. Additionally, ClickOS is missing important functionality that OS^v has, such as support for SMP guests and a TCP stack.

6 Conclusions and Future Work

We have shown that OS^v is, in many respects, a more suitable operating system for virtual machines in the cloud than are traditional operating systems such as Linux. OS^v outperforms Linux in many benchmarks, it makes for small images, and its boot time is barely noticeable. OS^v is a young project, and we believe that with continued work we can further improve its performance.

While OS^v improves the performance of existing applications, some of the most dramatic improvements we've seen came from adding non-POSIX API to OS^v. For example, the shrinker API allows an OS^v-aware application to make better use of available memory, and a packet-filtering APIs reduces the overheads of the standard socket APIs. We plan to continue to explore new interfaces to add to OS^v to further improve application performance. Areas of exploration will include network APIs and cooperative scheduling.

Instead of modifying many individual applications, a promising future direction is to modify a runtime environment, such as the JVM, on which many applications run. This will allow us to run unmodified applications, while still benefiting from new OS^v APIs. The JVM balloon we presented is an example of this direction.

Finally, we hope that the availability of OS^v, with its small modern code and broad usability (not limited to specific languages, hypervisors or applications) will encourage more research on operating systems for VMs.

7 Acknowledgments

Being an Open Source project, we would like to thank our community contributors, especially those ones working as volunteers.

8 Availability

OS^v is BSD-licensed open source, available at:

`https://github.com/cloudius-systems/osv`

More information is available at `http://osv.io/`.

References

[1] AMMONS, G., APPAVOO, J., BUTRICO, M., DA SILVA, D., GROVE, D., KAWACHIYA, K., KRIEGER, O., ROSENBURG, B., VAN HENSBERGEN, E., AND WISNIEWSKI, R. W. Libra: a library operating system for a JVM in a virtualized execution environment. In *Proceedings of the 3rd international conference on Virtual execution environments* (2007), ACM, pp. 44–54.

[2] ANDERSON, T. E. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems 1*, 1 (1990), 6–16.

[3] DES LIGNERIS, B. Virtualization of Linux based computers: the Linux-VServer project. In *International Symposium on High Performance Computing Systems and Applications* (2005), IEEE, pp. 340–346.

[4] DOUCEUR, J. R., ELSON, J., HOWELL, J., AND LORCH, J. R. Leveraging legacy code to deploy desktop applications on the web. In *OSDI* (2008), pp. 339–354.

[5] ENGLER, D. R., KAASHOEK, M. F., ET AL. Exokernel: An operating system architecture for application-level resource management. In *ACM SIGOPS Operating Systems Review* (1995), vol. 29, pp. 251–266.

[6] FITZPATRICK, B. Distributed caching with memcached. *Linux Journal*, 124 (2004).

[7] FRIEBEL, T., AND BIEMUELLER, S. How to deal with lock holder preemption. *Xen Summit North America* (2008).

[8] GIDENSTAM, A., AND PAPATRIANTAFILOU, M. LFthreads: A lock-free thread library. In *Principles of Distributed Systems*. Springer, 2007, pp. 217–231.

[9] HOWELL, J., PARNO, B., AND DOUCEUR, J. R. How to run POSIX apps in a minimal picoprocess. In *2013 USENIX ATC* (2013), pp. 321–332.

[10] JACOBSON, V., AND FELDERMAN, R. Speeding up networking. *Linux Conference Australia* (2006).

[11] JONES, R. A. A network performance benchmark (revision 2.0). Tech. rep., Hewlett Packard, 1995.

[12] KLEIMAN, S. R. Vnodes: An architecture for multiple file system types in Sun UNIX. In *USENIX Summer* (1986), vol. 86, pp. 238–247.

[13] LIANG, S. *The Java Native Interface: Programmer's Guide and Specification*. Addison-Wesley, 1999.

[14] MADHAVAPEDDY, A., MORTIER, R., ROTSOS, C., SCOTT, D., SINGH, B., GAZAGNAIRE, T., SMITH, S., HAND, S., AND CROWCROFT, J. Unikernels: Library operating systems for the cloud. In *ASPLOS* (2013), ACM.

[15] MARTINS, J., AHMED, M., RAICIU, C., OLTEANU, V., HONDA, M., BIFULCO, R., AND HUICI, F. ClickOS and the art of network function virtualization. In *USENIX NSDI* (2011).

[16] MASSALIN, H., AND PU, C. A lock-free multiprocessor OS kernel. *ACM SIGOPS Operating Systems Review 26*, 2 (1992), 108.

[17] MCKENNEY, P. E., AND SLINGWINE, J. D. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems* (1998), pp. 509–518.

[18] MEGIDDO, N., AND MODHA, D. Outperforming LRU with an adaptive replacement cache algorithm. *Computer 37*, 4 (April 2004), 58–65.

[19] MICHAEL, M. M., AND SCOTT, M. L. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. *Journal of Parallel and Distributed Computing 51*, 1 (1998), 1–26.

[20] NISHTALA, R., FUGAL, H., GRIMM, S., KWIATKOWSKI, M., LEE, H., LI, H. C., MCELROY, R., PALECZNY, M., PEEK, D., SAAB, P., ET AL. Scaling Memcache at Facebook. In *USENIX NSDI* (2013).

[21] PORTER, D. E., BOYD-WICKIZER, S., HOWELL, J., OLINSKY, R., AND HUNT, G. C. Rethinking the library OS from the top down. *ACM SIGPLAN Notices 46*, 3 (2011), 291–304.

[22] PRINTEZIS, T. Garbage collection in the Java HotSpot virtual machine, 2005.

[23] RIZZO, L. netmap: a novel framework for fast packet I/O. In *USENIX ATC* (2012).

[24] ROMER, T. H., OHLRICH, W. H., KARLIN, A. R., AND BERSHAD, B. N. Reducing TLB and memory overhead using online superpage promotion. In *Computer Architecture, 1995. Proceedings., 22nd Annual International Symposium on* (1995), IEEE, pp. 176–187.

[25] RUSSELL, R. virtio: towards a de-facto standard for virtual I/O devices. 95–103.

[26] SOLTESZ, S., PÖTZL, H., FIUCZYNSKI, M. E., BAVIER, A., AND PETERSON, L. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. In *ACM SIGOPS Operating Systems Review* (2007), vol. 41, ACM, pp. 275–287.

[27] TENE, G., IYENGAR, B., AND WOLF, M. C4: the continuously concurrent compacting collector. *ACM SIGPLAN Notices 46*, 11 (2011), 79–88.

[28] UHLIG, V., LEVASSEUR, J., SKOGLUND, E., AND DANNOWSKI, U. Towards scalable multiprocessor virtual machines. In *Virtual Machine Research and Technology Symposium* (2004), pp. 43–56.

[29] VMWARE INC. ESX Server 2 - architecture and performance implications. Tech. rep., VMWare, 2005.

[30] WALDSPURGER, C. A. Memory resource management in VMware ESX server. *SIGOPS Oper. Syst. Rev. 36*, SI (Dec. 2002), 181–194.

[31] ZHAO, W., WANG, Z., AND LUO, Y. Dynamic memory balancing for virtual machines. *ACM SIGOPS Operating Systems Review 43*, 3 (2009), 37–47.