My VM is Lighter (and Safer) than your Container

Filipe Manco
NEC Laboratories Europe
filipe.manco@gmail.com

Costin Lupu
Univ. Politehnica of Bucharest
costin.lupu@cs.pub.ro

Florian Schmidt
NEC Laboratories Europe
florian.schmidt@neclab.eu

Jose Mendes
NEC Laboratories Europe
jose.mendes@neclab.eu

Simon Kuenzer
NEC Laboratories Europe
simon.kuenzer@neclab.eu

Sumit Sati
NEC Laboratories Europe
sati.vicky@gmail.com

Kenichi Yasukata
NEC Laboratories Europe
kenichi.yasukata@neclab.eu

Costin Raiciu
Univ. Politehnica of Bucharest
costin.raiciu@cs.pub.ro

Felipe Huici
NEC Laboratories Europe
felipe.huici@neclab.eu

ABSTRACT

Containers are in great demand because they are lightweight when compared to virtual machines. On the downside, containers offer weaker isolation than VMs, to the point where people run containers in virtual machines to achieve proper isolation. In this paper, we examine whether there is indeed a strict tradeoff between isolation (VMs) and efficiency (containers). We find that VMs can be as nimble as containers, as long as they are small and the toolstack is fast enough.

We achieve lightweight VMs by using unikernels for specialized applications and with Tinyx, a tool that enables creating tailor-made, trimmed-down Linux virtual machines. By themselves, lightweight virtual machines are not enough to ensure good performance since the virtualization control plane (the toolstack) becomes the performance bottleneck. We present LightVM, a new virtualization solution based on Xen that is optimized to offer fast boot-times regardless of the number of active VMs. LightVM features a complete redesign of Xen's control plane, transforming its centralized operation to a distributed one where interactions with the hypervisor are reduced to a minimum. LightVM can boot a VM in 2.3ms, comparable to `fork/exec` on Linux (1ms), and two orders of magnitude faster than Docker. LightVM can pack thousands of LightVM guests on modest hardware with memory and CPU usage comparable to that of processes.

This work is licensed under a Creative Commons Attribution International 4.0 License.

SOSP '17, October 28, 2017, Shanghai, China
© 2017 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-5085-3/17/10.
https://doi.org/10.1145/3132747.3132763

CCS CONCEPTS

• **Software and its engineering → Virtual machines**; *Operating Systems*;

KEYWORDS

Virtualization, unikernels, specialization, operating systems, Xen, containers, hypervisor, virtual machine.

ACM Reference Format:
Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. 2017. My VM is Lighter (and Safer) than your Container. In *Proceedings of SOSP '17: ACM SIGOPS 26th Symposium on Operating Systems Principles, Shanghai, China, October 28, 2017 (SOSP '17),* 16 pages.
https://doi.org/10.1145/3132747.3132763

1 INTRODUCTION

Lightweight virtualization technologies such as Docker [6] and LXC [25] are gaining enormous traction. Google, for instance, is reported to run all of its services in containers [4], and Container as a Service (CaaS) products are available from a number of major players including Azure's Container Service [32], Amazon's EC2 Container Service and Lambda offerings [1, 2], and Google's Container Engine service [10].

Beyond these services, lightweight virtualization is crucial to a wide range of use cases, including just-in-time instantiation of services [23, 26] (e.g., filters against DDoS attacks, TCP acceleration proxies, content caches, etc.) and NFV [41, 51], all while providing significant cost reduction through consolidation and power minimization [46].

The reasons for containers to have taken the virtualization market by storm are clear. In contrast to heavyweight, hypervisor-based technologies such as VMWare, KVM or Xen, they provide extremely fast instantiation times, small per-instance memory footprints, and high density on a single host, among other features.

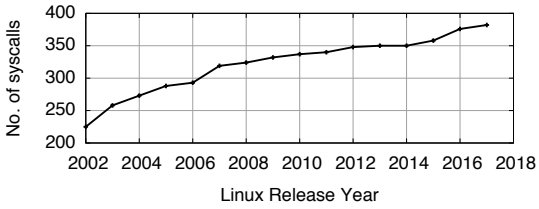


Figure 1: The unrelenting growth of the Linux syscall API over the years (x86_32) underlines the difficulty of securing containers.

However, no technology is perfect, and containers are no exception: security is a continuous thorn in their side. The main culprit is the hugely powerful kernel syscall API that containers use to interact with the host OS. This API is very broad as it offers kernel support for process and thread management, memory, network, filesystems, IPC, and so forth: Linux, for instance, has 400 different system calls [37], most with multiple parameters and many with overlapping functionality; moreover, the number of syscalls is constantly increasing (see figure 1). The syscall API is fundamentally more difficult to secure than the relatively simple x86 ABI offered by virtual machines where memory isolation (with hardware support) and CPU protection rings are sufficient. Despite the many isolation mechanisms introduced in the past few years, such as process and network namespaces, root jails, seccomp, etc, containers are the target of an ever increasing number of exploits [11, 22]. To complicate matters, any container that can monopolize or exhaust system resources (e.g., memory, file descriptors, user IDs, forkbombs) will cause a DoS attack on all other containers on that host [14, 35].

At least for multi-tenant deployments, this leaves us with a difficult choice between (1) containers and the security issues surrounding them and (2) the burden coming from heavyweight, VM-based platforms. Securing containers in the context of an ever-expanding and powerful syscall API is elusive at best. Could we make virtualization faster and more nimble, much like containers? The explicit goal would be to achieve performance in the same ball-park as containers: instantiation in milliseconds, instance memory footprints of a few MBs or less, and the ability to concurrently run one thousand or more instances on a single host.

In this paper we introduce LightVM, a *lightweight* virtualization system based on a type-1 hypervisor. LightVM retains the strong isolation virtual machines are well-known for while providing the performance characteristics that make containers such an attractive proposition. In particular, we make the following contributions:

- An analysis of the performance bottlenecks preventing traditional virtualization systems from achieving container-like dynamics (we focus our work on Xen).
- An overhaul of Xen's architecture, completely removing its back-end registry (a.k.a. the XenStore), which constitutes a performance bottleneck. We call this `noxs` (no XenStore), and its implementation results in significant improvements for boot and migration times, among other metrics.
- A revamp of Xen's toolstack, including a number of optimizations and the introduction of a *split* toolstack that separates functionality that can be run periodically, offline, from that which must be carried out when a command (e.g., VM creation) is issued.
- The development of Tinyx, an automated system for building minimalistic Linux-based VMs, as well as the development of a number of unikernels. These lightweight VMs are fundamental to achieving high performance numbers, but also for discovering performance bottlenecks in the underlying virtualization platform.
- A prototypical implementation along with an extensive performance evaluation showing that LightVM is able to boot a (unikernel) VM in as little as 2.3ms, reach same-host VM densities of up to 8000 VMs, migration and suspend/resume times of 60ms and 30ms/25ms respectively, and per-VM memory footprints of as little as 480KB (on disk) and 3.6MB (running).

To show its applicability, we use LightVM to implement four use cases: personalized firewalls, just-in-time service instantiation, high density TLS termination and a lightweight compute service akin to Amazon Lambda or Google's Cloud Functions but based on a Python unikernel. LightVM is available as open source at http://sysml.neclab.eu/projects/lightvm .

2 REQUIREMENTS

The goal is to be able to provide lightweight virtualization on top of hypervisor technology. More specifically, as requirements, we are interested in a number of characteristics typical of containers:

- **Fast Instantiation:** Containers are well-known for their small startup times, frequently in the range of hundreds of milliseconds or less. In contrast, virtual machines are infamous for boot times in the range of seconds or longer.
- **High Instance Density:** It is common to speak of running hundreds or even up to a thousand containers on a single host, with people even pushing this boundary up to 10,000 containers [17]. This is much higher than what VMs can typically achieve, which lies more in the range of tens or hundreds at most, and normally requires fairly powerful and expensive servers.
- **Pause/unpause:** Along with short instantiation times, containers can be paused and unpaused quickly. This

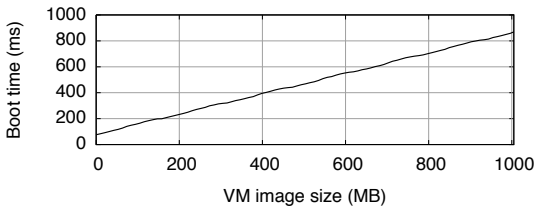


Figure 2: Boot times grow linearly with VM image size.

can be used to achieve even higher density by pausing idle instances, and more generally to make better use of CPU resources. Amazon Lambda, for instance, "freezes" and "thaws" containers.

The single biggest factor limiting both the scalability and performance of virtualization is the size of the guest virtual machines: for instance, both the on-disk image size as well as the running memory footprint are on the order of hundreds of megabytes to several gigabytes for most Linux distributions. VM memory consumption imposes a hard upper bound on the number of instances that can be run on the same server. Containers typically require much less memory than virtual machines (a few MBs or tens of MBs) because they share the kernel and have smaller root filesystems.

Large VMs also slow down instantiation times: the time needed to read the image from storage, parse it and lay it out in memory grows linearly with image size. This effect is clearly shown in Figure 2 where we boot the same unikernel VM from images of different sizes, all stored in a ramdisk. We increase the size by injecting binary objects into the uncompressed image file. This ensures that the results are due to the effects that image size has on VM initialization.

3 LIGHTWEIGHT VMS

The first step towards achieving our goals is to reduce both the image size and the memory footprint of virtual machines. We observe, as others [27], that most containers and virtual machines run a single application; if we reduce the functionality of the VM to include only what is necessary for that application, we expect to reduce the memory usage dramatically. Concretely, we explore two avenues to optimize virtual machine images:

- **Unikernels**: tiny virtual machines where a minimalistic operating system (such as MiniOS [34]) is linked directly with the target application. The resulting VM is typically only a few megabytes in size and can only run the target application; examples include ClickOS [29] and Mirage [27].
- **Tinyx**: a tool that we have built to create a tiny Linux distribution around a specified application. This results in images that are a few tens of MBs in size and need around 30MBs of RAM to boot.

3.1 Unikernels

There is a lot of prior work showing that unikernels have very low memory footprint, and for specific applications there already exist images that one can re-use: ClickOS is one such example that can run custom Click modular router configurations composed of known elements. Mirage [27] is another example that takes applications written in OCaml and creates a minimalistic app+OS combo that is packed as a guest VM.

If one needs to create a new unikernel, the simplest is to rely on Mini-OS [34], a toy guest operating system distributed with Xen: its functionality is very limited, there is no user/kernel separation and no processes/fork. For instance, only 50 LoC are needed to implement a TCP server over Mini-OS that returns the current time whenever it receives a connection (we also linked the lwip networking stack). The resulting VM image, which we will refer to as the daytime unikernel, is only 480KB (uncompressed), and can run in as little as 3.6MB of RAM.[1] We use the daytime unikernel as a lower bound of memory consumption for possible VMs.

We have also created unikernels for more interesting applications, including a TLS termination proxy and Minipython, a Micropython-based unikernel to be used by Amazon lambda-like services; both have images of around 1MB and can run with just 8MB of memory.

In general, though, linking existing applications that rely on the Linux syscall API to Mini-OS is fairly cumbersome and requires a lot of expert time. That is why we also explored another approach to creating lightweight VMs based on the Linux kernel, described next.

3.2 Tinyx

Tinyx is an automated build system that creates minimalistic Linux VM images targeted at running a single application (although the system supports having multiple ones). The tool builds, in essence, a VM consisting of a minimalistic, Linux-based distribution along with an optimized Linux kernel. It provides a middle point between a highly specialized unikernel, which has the best performance but requires porting of applications to a minimalistic OS, and a full-fledged general-purpose OS VM that supports a large number of applications out of the box but incurs performance overheads.

The Tinyx build system takes two inputs: an application to build the image for (e.g., nginx) and the platform the image will be running on (e.g., a Xen VM). The system separately builds a filesystem/distribution and the kernel itself. For the

[1]The 3.6MB requires a small patch to Xen's toolstack to get around the fact that it imposes a 4MB minimum by default.

distribution, Tinyx includes the application, its dependencies, and BusyBox (to support basic functionality).

To derive dependencies, Tinyx uses (1) `objdump` to generate a list of libraries and (2) the Debian package manager. To optimize the latter, Tinyx includes a blacklist of packages that are marked as required (mostly for installation, e.g., `dpkg`) but not strictly needed for running the application. In addition, we include a whitelist of packages that the user might want to include irrespective of dependency analysis.

Tinyx does not directly create its images from the packages since the packages include installation scripts which expect utilities that might not be available in the minimalistic Tinyx distribution. Instead, Tinyx first mounts an empty OverlayFS directory over a Debian minimal debootstrap system. In this mounted directory we install the minimal set of packages discovered earlier as would be normally done in Debian. Since this is done on an overlay mounted system, unmounting this overlay gives us all the files which are properly configured as they would be on a Debian system. Before unmounting, we remove all cache files, any dpkg/apt related files, and other unnecessary directories. Once this is done, we overlay this directory on top of a BusyBox image as an underlay and take the contents of the merged directory; this ensures a minimalistic, application-specific Tinyx "distribution". As a final step, the system adds a small glue to run the application from BusyBox's init.

To build the kernel, Tinyx begins with the "tinyconfig" Linux kernel build target as a baseline, and adds a set of built-in options depending on the target system (e.g., Xen or KVM support); this generates a working kernel image. By default, Tinyx disables module support as well as kernel options that are not necessary for virtualized systems (e.g., baremetal drivers). Optionally, the build system can take a set of user-provided kernel options, disable each one in turn, rebuild the kernel with the `olddefconfig` target, boot the Tinyx image, and run a user-provided test to see if the system still works (e.g., in the case of an nginx Tinyx image, the test includes attempting to wget a file from the server); if the test fails, the option is re-enabled, otherwise it is left out of the configuration. Combined, all these heuristics help Tinyx create kernel images that are half the size of typical Debian kernels and significantly smaller runtime memory usage (1.6MB for Tinyx vs. 8MB for the Debian we tested).

4 VIRTUALIZATION TODAY

Armed with our tiny VM images, we are now ready to explore the performance of existing virtualization technologies. We base our analysis on Xen [3], which is a type-1 hypervisor widely used in production (e.g., in Amazon EC2). Xen has a small trusted computing base and its code is fairly mature, resulting in strong isolation (the ARM version of the hypervisor, for instance, consists of just 11.4K LoC [43], and dissagregation [5] can be used to keep the size of critical `Dom0` code low). The competing hypervisor, KVM, is based on the Linux kernel and has a much larger trusted computing base. To better understand the following investigation, we start with a short introduction on Xen.

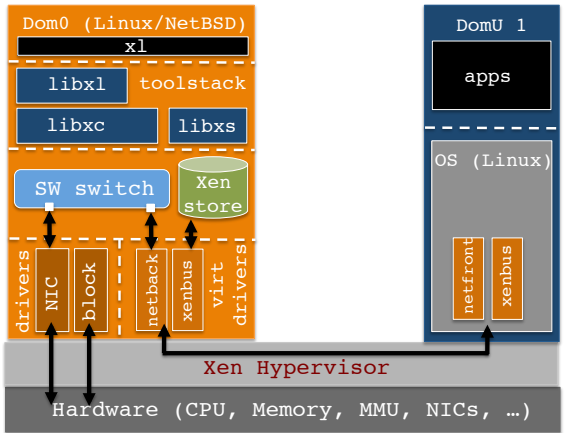


Figure 3: The Xen architecture including toolstack, the XenStore, software switch and split drivers between the driver domain (Dom0) and the guests (DomUs).

4.1 Short Xen Primer

The Xen hypervisor only manages basic resources such as CPUs and memory (see Figure 3). When it finishes booting, it automatically creates a special virtual machine called `Dom0`. `Dom0` typically runs Linux and hosts the *toolstack*, which includes the `xl` command and the `libxl` and `libxc` libraries needed to carry out commands such as VM creation, migration and shutdown.

`Dom0` also hosts the XenStore, a `proc`-like central registry that keeps track of management information such as which VMs are running and information about their devices, along with the `libxs` library containing code to interact with it. The XenStore provides *watches* that can be associated with particular directories of the store and that will trigger callbacks whenever those directories are modified.

Typically, `Dom0` also hosts a software switch (Open vSwitch is the default) to mux/demux packets between NICs and the VMs, as well as the (Linux) drivers for the physical devices.[2] For communication between `Dom0` and the other guests, Xen implements a *split-driver* model: a virtual back-end driver running in `Dom0` (e.g., the netback driver for networking) communicates over shared memory with a front-end driver running in the guests (the netfront driver). So-called *event*

[2]Strictly speaking, this functionality can be put in a separate VM called a *driver domain*, but in most deployments `Dom0` acts as a driver domain.

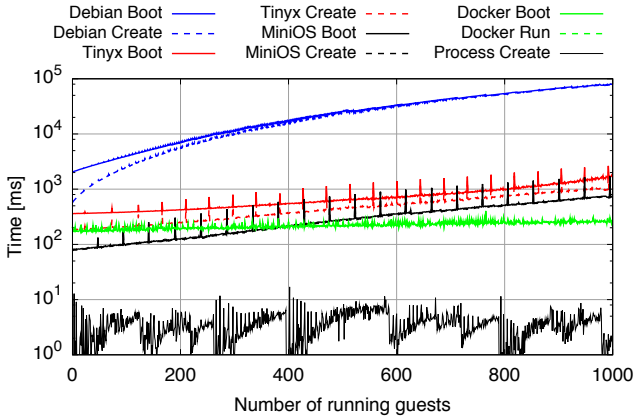


Figure 4: Comparison of domain instantiation and boot times for several guest types. With small guests, instantiation accounts for most of the delay when bringing up a new VM.

channels, essentially software interrupts, are used to notify drivers about the availability of data.

4.2 Overhead Investigation

As a testing environment, we use a machine with an Intel Xeon E5-1630 v3 CPU at 3.7 GHz and 128 GiB of DDR4 memory, and Xen and Linux versions 4.8. We then sequentially start 1000 virtual machines and measure the time it takes to *create* each VM (the time needed for the toolstack to prepare the VM), and the time it takes the VM to *boot*. We do this for three types of VMs that exemplify vastly different sizes: first, a VM running a minimal install of Debian jessie that we view as a typical VM used in practice; second, Tinyx, where the distribution is bundled into the kernel image as an initramfs; and finally, the daytime unikernel.

Figure 4 shows creation and boot times for these three VM types, Docker containers, and basic Linux processes (as a baseline). All VM images are stored on a ramdisk to eliminate the effects that disk I/O would have on performance.

The Debian VM is 1.1GB in size; it takes Xen around 500ms to create the VM when there are no other VMs running, and it takes the VM 1.5 seconds to boot. The Tinyx VM (9.5MB image) is created in 360ms and needs a further 180ms to boot. The first unikernel (480KB image) is created in 80ms, and needs 3ms to boot.

Docker containers start in around 200ms, and a process is created and launched (using fork/exec) in 3.5ms on average (9ms at the 90% percentile). However, for both processes and containers creation time does not depend on the number of existing containers or processes.

As we keep creating VMs, however, the creation time increases noticeably (note the logarithmic scale): it takes 42s, 10s and 700ms to create the thousandth Debian, Tinyx, and unikernel guest, respectively. These results are surprising, since all the VMs are idle after they boot, so the total system utilization should be low regardless of the number of VMs. Another result is also apparent from this test: as the size of the VM decreases, the creation time contributes a larger and larger fraction of the time that it takes from starting the VM creation to its availability: with lightweight VMs, the instantiation of new VMs becomes the main bottleneck.

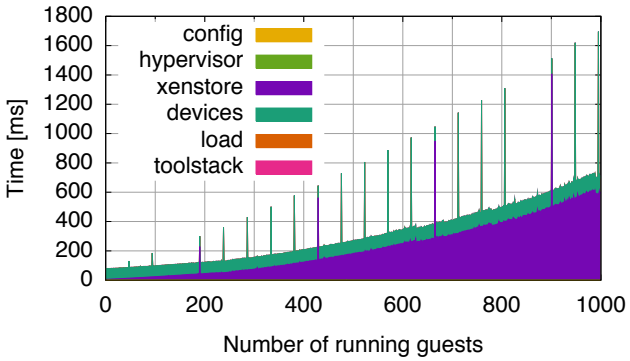


Figure 5: Breakdown of the VM creation overheads shows that the main contributors are interactions with the XenStore and the creation of virtual devices.

To understand VM creation overheads, we instrumented Xen's `xl` command-line tool and its `libxl` library, and categorized the work done into several categories:

- Parsing the configuration file that describes the VM (kernel image, virtual network/block devices, etc.).
- Interacting with the hypervisor to, for example, reserve and prepare the memory for the new guest, create the virtual CPUs, and so on.
- Writing information about the new guest in the XenStore, such as its name.
- Creating and configuring the virtual devices.
- Parsing the kernel image and loading it into memory.
- Internal information and state keeping of the toolstack that do not fit into any of the above categories.

Figure 5 shows the creation overhead categorized in this way. It is immediately apparent that there are two main contributors to the VM creation overhead: the XenStore interaction and the device creation, to the point of negligibility of all other categories.[3] Device creation dominates the guest instantiation times when the number of currently running guests is low; its overhead stays roughly constant when we keep adding virtual machines.

[3]Note that this already uses `oxenstored`, the faster of the two available implementations of the XenStore. Results with `cxenstored` show much higher overheads.

However, the time spent on XenStore interactions increases superlinearly, for several reasons. The protocol used by the XenStore is quite expensive, where each operation requires sending a message and receiving an acknowledgment, each triggering a software interrupt: a single read or write thus triggers at least two, and most often four, software interrupts and multiple domain changes between the guest, hypervisor and `Dom0` kernel and userspace; as we increase the number of VMs, so does the load on this protocol. Secondly, writing certain types of information, such as unique guest names, incurs overhead linear with the number of machines because the XenStore compares the new entry against the names of all other already-running guests before accepting the new guest's name. Finally, some information, such as device initialization, requires writing data in multiple XenStore records where atomicity is ensured via transactions. As the load increases, XenStore interactions belonging to different transactions frequently overlap, resulting in failed transactions that need to be retried.

Finally, it is worth noting that the spikes on the graph are due to the fact that the XenStore logs every access to log files (20 of them), and rotates them when a certain maximum number of lines is reached (13,215 lines by default); the spikes happen when this rotation takes place. While disabling this logging would remove the spikes, it would not help in improving the overall creation times, as we verified in further experiments not shown here.

5 LIGHTVM

Our target is to achieve VM boot times comparable to process startup times. Xen has not been engineered for this objective, as the results in the previous section show, and the root of these problems is deeper than just inefficient code. For instance, one fundamental problem with the XenStore is its centralized, filesystem-like API which is simply too slow for use during VM creation and boot, requiring tens of interrupts and privilege domain crossings. Contrast this to the `fork` system call which requires a single software interrupt – a single user-kernel crossing. To achieve millisecond boot times we need much more than simple optimizations to the existing Xen codebase.

To this end, we introduce LightVM, a complete re-design of the basic Xen control plane optimized to provide lightweight virtualization. The architecture of LightVM is shown in Figure 6. LightVM does not use the XenStore for VM creation or boot anymore, using instead a lean driver called `noxs` that addresses the scalability problems of the XenStore by enabling direct communication between the frontend and backend drivers via shared memory instead of relaying messages through the XenStore. Because `noxs` does not rely on a message passing protocol but rather on shared pages

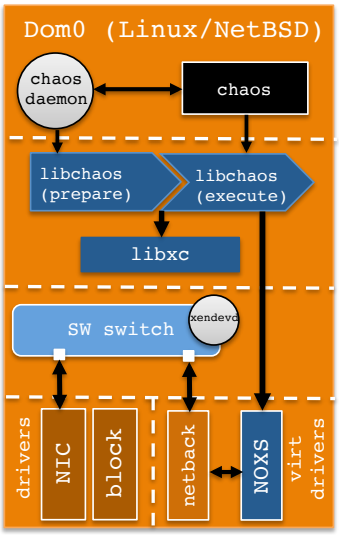


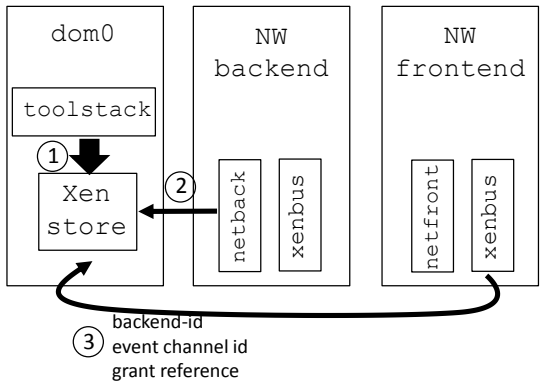
Figure 6: LightVM architecture showing `noxs`, `xl`'s replacement (`chaos`), the split toolstack and accompanying daemon, and `xendevd` in charge of quickly adding virtual interfaces to the software switch.

mapped in the guest's address space, reducing the number of software interrupts and domain crossings needed for VM operations (create/save/resume/migrate/destroy).

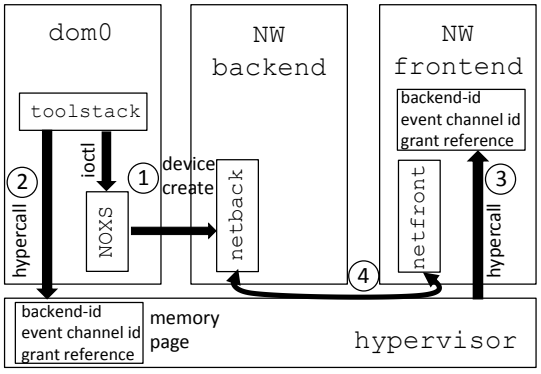
LightVM provides a split toolstack that separates VM creation functionality into a prepare and an execute phase, reducing the amount of work to be done on VM creation. We have also implemented `chaos/libchaos`, a new virtualization toolstack that is much leaner than the standard `xl/libxl`, in addition to a small daemon called `xendevd` that quickly adds virtual interfaces to the software switch or handles block device images' setup. We cover each of these in detail in the next sections.

5.1 Noxs (no XenStore) and the Chaos Toolstack

The XenStore is crucial to the way Xen functions, with many `xl` commands making heavy use of it. By way of illustration, Figure 7a shows the process when creating a VM and its (virtual) network device. First, the toolstack writes an entry to the network back-end's directory, essentially announcing the existence of a new VM in need of a network device. Previous to that, the back-end placed a watch on that directory; the toolstack writing to this directory triggers the back-end to assign an event channel and other information (e.g., grant references, a mechanism for sharing memory between guests) and to write it back to the XenStore (step 2 in the figure). Finally, when the VM boots up it contacts the XenStore to retrieve the information previously written by



(a) XenStore



(b) noxs

Figure 7: Standard VM creation process in Xen using the XenStore versus our noxs implementation.

the network back-end (step 3). The above is a simplification: in actuality, the VM creation process alone can require interaction with over 30 XenStore entries, a problem that is exacerbated with increasing number of VMs and devices.

Is it possible to forego the use of the XenStore for operations such as creation, pause/unpause and migration? As it turns out, most of the necessary information about a VM is already kept by the hypervisor (e.g., the VM's id, but not the name, which is kept in the XenStore but is not needed during boot). The insight here is that the hypervisor already acts as a sort of centralized store, so we can extend its functionality to implement our noxs (no XenStore) mechanism.

Specifically, we begin by replacing `libxl` and the corresponding `xl` command with a streamlined, thin library and command called `libchaos` and `chaos`, respectively (cf. Figure 6); these no longer make use of the XenStore and its accompanying `libxs` library.

In addition, we modify Xen's hypervisor to create a new, special device memory page for each new VM that we use to keep track of a VM's information about any devices, such as block and networking, that it may have. We also include a hypercall to write to and read from this memory page, and make sure that, for security reasons, the page is shared read-only with guests, with only `Dom0` allowed to request modifications.

When a `chaos` create command is issued, the toolstack first requests the creation of devices from the back-end(s) through an `ioctl` handled by the `noxs` Linux kernel module (step 1 in Figure 7b).[4] The back-end then returns the details about the communication channel for the front-end. Second, the toolstack calls the new hypercall asking the hypervisor to add these details to the device page (step 2).

When the VM boots, instead of contacting the XenStore, it will ask the hypervisor for the address of the device page and will map the page into its address space using hypercalls (step 3 in the figure); this requires modifications to the guest's operating system, which we have done for Linux and Mini-OS. The guest will then use the information in the page to initiate communication with the back-end(s) by mapping the grant and binding to the event channel (step 4). At this stage, the front and back-ends set up the device by exchanging information such as its state and its MAC address (for networking); this information was previously kept in the XenStore and is now stored in a device control page pointed to by the grant reference. Finally, front and back-ends notify each other of events through the event channel, which replaces the use of XenStore watches.

To support migration without a XenStore, we create a new pseudo-device called `sysctl` to handle power-related operations and implement it following Xen's split driver model, with a back-end driver (sysctlback) and a front-end (sysctlfront) one. These two drivers share a device page through which communication happens and an event channel.

With this in place, migration begins by `chaos` opening a TCP connection to a migration daemon running on the remote host and by sending the guest's configuration so that the daemon pre-creates the domain and creates the devices. Next, to suspend the guest, `chaos` issues an `ioctl` to the sysctl back-end, which will set a field in the shared page to denote that the shutdown reason is suspend, and triggers the event channel. The front-end will receive the request to shutdown, upon which the guest will save its internal state and unbind `noxs`-related event channels and device pages. Once the guest is suspended we rely on `libxc` code to send the guest data to the remote host.

5.2 Split Toolstack

In the previous section we showed that a large fraction of the overheads related to VM creation and other operations comes from the toolstack itself. Upon closer investigation, it

[4]Currently this mechanism only works if the back-ends run in `Dom0`, but the architecture allows for back-ends to run on a different virtual machine.

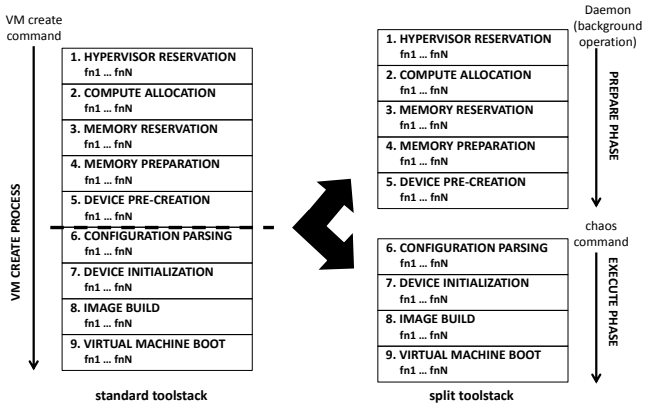


Figure 8: Toolstack split between functionality belonging to the prepare phase, carried out periodically by the chaos daemon, and an execute phase, directly called by chaos when a command is issued.

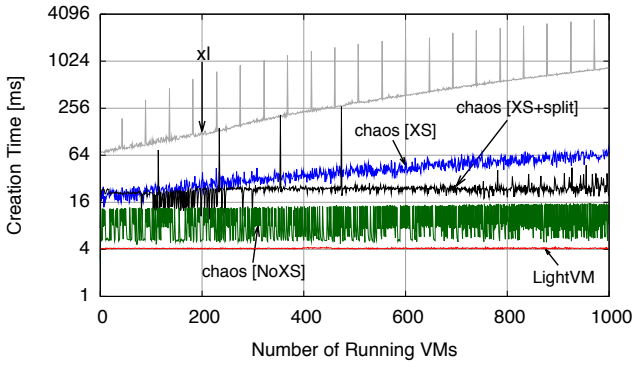


Figure 9: Creation times for up to 1,000 instances of the daytime unikernel for all combinations of LightVM's mechanisms. "xl" denotes standard Xen with no optimizations.

turns out that a significant portion of the code that executes when, for instance, a VM create command is issued, does not actually need to run at VM creation time. This is because this code is *common to all VMs*, or at least common to all VMs with similar configurations. For example, the amount of memory may differ between VMs, but there will generally only be a small number of differing memory configurations, similar to OpenStack's flavors. This means that VMs can be pre-executed and thus off-loaded from the creation process.

To take advantage of this, we replace the standard Xen toolstack with the `libchaos` library and split it into two phases. The *prepare* phase (see Figure 8) is responsible for functionality common to all VMs such as having the hypervisor generate an ID and other management information and allocating CPU resources to the VM. We *offload* this functionality to the `chaos` daemon, which generates a number of VM *shells* and places them in a pool. The daemon ensures that there is always a certain (configurable) number of shells available in the system.

The *execute* phase then begins when a VM creation command is issued. First, `chaos` parses the configuration file for the VM to be created. It then contacts the daemon and asks for a shell fitting the VM requirements, which is then removed from the pool. On this shell, the remaining VM-specific operations, such as loading the kernel image into memory and finalizing the device initialization, are executed to create the VM, which is then booted.

5.3 Replacing the Hotplug Script: xendevd

The creation of a virtual device by the driver domain usually requires some mechanism to setup the device in user-space (e.g., by adding a *vif* to the bridge). With standard Xen this process is done either by `xl`, calling bash scripts that take care of the necessary initialization or by `udevd`, calling the same scripts when the backend triggers the udev event. The script that is executed is usually user-configured, giving great flexibility to implement different scenarios. However launching and executing bash scripts is a slow process taking tens of milliseconds, considerably slowing down the boot process. To work around this, we implemented this mechanism as a binary daemon called *xendevd* that listens for udev events from the backends and executes a pre-defined setup without forking or bash scripts, reducing setup time.

6 EVALUATION

In this section we present a performance evaluation of LightVM, including comparisons to standard Xen and, where applicable, Docker containers. We use two x86 machines in our tests: one with an Intel Xeon E5-1630 v3 CPU at 3.7 GHz (4 cores) and 128GB of DDR4 RAM, and another one consisting of four AMD Opteron 6376 CPUs at 2.3 GHz (with 16 cores each) and 128GB of DDR3 RAM. Both servers run Xen 4.8.

We use a number of different guests: (1) three Mini-OS-based unikernels, including `noop`, the daytime unikernel, and one based on Micropython [31] which we call Minipython; (2) a Tinyx `noop` image (no apps installed) and a Tinyx image with Micropython; and (3) a Debian VM. For container experiments we use Docker version 1.13.

6.1 Instantiation Times

We want to measure how long it takes to create and boot a virtual machine, how that scales as the number of running VMs on the system increases, and how both of these compare to containers. We further want to understand how the LightVM mechanisms affect these times.

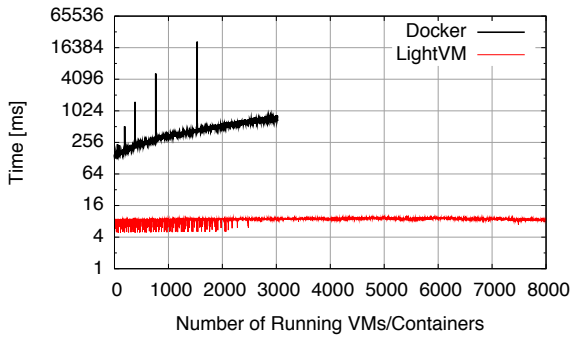


Figure 10: LightVM boot times on a 64-core machine versus Docker containers.

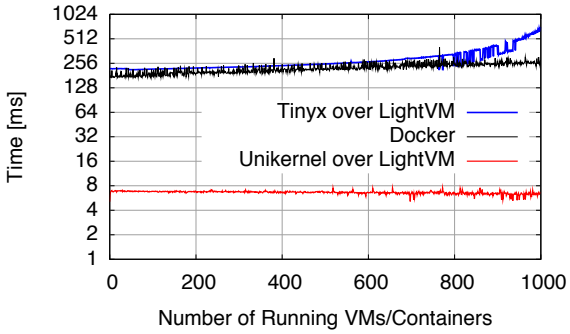


Figure 11: Boot times for unikernel and Tinyx guests versus Docker containers.

For the first test, we boot up to 1,000 daytime unikernels, and we measure the time it takes for the n'th unikernel to be created. We repeat this experiment with all combinations of the LightVM mechanisms: `chaos` + XenStore, `chaos` + `noxs`, `chaos` + split toolstack and `chaos` + `noxs` + split toolstack; we also include measurements when running out-of-the-box Xen (labeled "xl"). We run the tests on the 4-core machine, with one core assigned to `Dom0` and the remaining three cores assigned to the VMs in a round-robin fashion.

The results are shown in Figure 9. Out of the box, Xen (the "xl" curve) has creation times of about 100ms for the first VM, scaling rather poorly up to a maximum of just under 1 second for the 1000th VM. In addition, the curve shows spikes at regular intervals as a result of the log rotation issue mentioned earlier in the paper. Replacing `xl` with `chaos` results in a noticeable improvement, with creation times ranging now from roughly 15 to 80ms. Adding the split toolstack mechanism to the equation improves scalability, showing a maximum of about 25ms for the last VMs. Removing the XenStore (`chaos` + `noxs` curve) provides great scalability, essentially yielding low creation times in the range of 8-15 ms for the last VMs. Finally, we obtain the best results with all of the optimizations (`chaos` + `noxs` + split toolstack) turned on: boot times as low as 4ms going up to just 4.1ms for the 1,000th VM. As a final point of reference, using a `noop` unikernel with no devices and all optimizations results in a minimum boot time of 2.3ms.

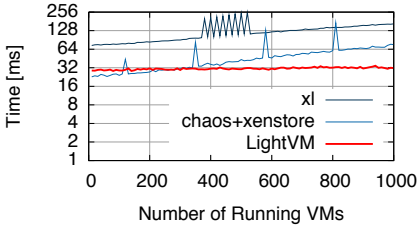
Next, we test LightVM against Docker containers when using even larger numbers of instances (see Figure 10). To measure this, we used the 64-core AMD machine, assigning 4 cores to `Dom0` and the remaining 60 to the VMs in a round-robin fashion; we used the `noop` unikernel for the guests themselves. As before, the best results come when using `chaos` + `noxs` + split toolstack, which shows good scalability with increasing number of VMs, up to 8,000 of them in this case. Docker containers start at about 150ms and ramp up to about 1 second for the 3,000th container. The spikes in that curve coincide with large jumps in memory consumption, and we stop at about 3,000 because after that the next large memory allocation consumes all available memory and the system becomes unresponsive.

For the final test we show how the boot times of a Tinyx guest compare to those of a unikernel, and we further plot a Docker container curve for comparison (see Figure 11). As expected, the unikernel performs best, but worthy of note is the fact that Tinyx, a Linux-based VM, performs rather close to Docker containers up to roughly 750 VMs (250 per core on our test machine). The increase in boot times as the number of VMs per core increases is due to the fact that even an idle, minimal Linux distribution such as Tinyx runs occasional background tasks. As the number of VMs per core increases, contention for CPU time increases, increasing the boot time of each VM. In contrast, idling docker containers or unikernels do not run such background tasks, leading to no noticeable increase in boot times.

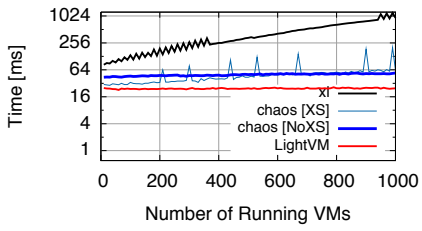
6.2 Checkpointing and Migration

In the next set of experiments we use the 4-core machine and the daytime unikernel to test save/restore (i.e., checkpointing) and migration times. In all tests we use a RAM disk for the filesystem so that the results are not skewed by hard drive access speeds. We assign two cores to `Dom0` and the remaining two to the VMs in a round-robin fashion.

For checkpointing, the experimental procedure is as follows. At every run of the test we start 10 guests and randomly pick 10 guests to be checkpointed; for instance, in the first run all 10 guests created are checkpointed, in the second one 20 guests exist out of which 10 are checkpointed, and so on up to a total of 1,000 guests. This is to show how quick checkpointing is when the system is already running N numbers of guests. The results are shown in Figure 12a and Figure 12b,



(a) Save



(b) Restore

Figure 12: Checkpointing times using the daytime unikernel.

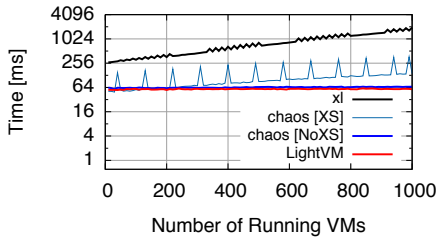


Figure 13: Migration times for the daytime unikernel.

split between save and restore times, respectively. The figures show that LightVM can save a VM in around 30ms and restore it in 20ms, regardless of the number of running guests, while standard Xen needs 128ms and 550ms respectively.

For migration tests we carry out the same procedure of starting 10 guests and randomly choosing 10 to migrate. Once the 10 guests are migrated, we replace the migrated guests with 10 new guests to make sure that the right number of guests are running on the source host for the next round of the test (e.g., if we have 100 guests, we migrate 10, leaving 90 on the source host; we then create 10, leaving 100 at the source host so that it is ready for the next round).

We plot migration times for the daytime unikernel in Figure 13. As in previous graphs, turning all optimizations on (`chaos`, `noxs` and split toolstack) yields the best results, with migration times of about 60ms irrespective of how many VMs are currently running on the host. For low number of VMs the `chaos` + XenStore slightly outperforms LightVM: this is due to device destruction times in `noxs` which we have not yet optimized and remain as future work.

6.3 Memory Footprint

One of the advantages of containers is that, since they use a common kernel, their memory usage stays low as their numbers increase. This is in contrast to VMs, where each instance is a full replica of the operating system and filesystem. In our next experiment we try to see if small VMs can get close to the memory scalability that containers provide. For the test we use the 4-core machine and allocate a single core to `Dom0` and the remaining 3 to the VMs as before. We use three types of guests: a Minipython unikernel, a Tinyx VM with Micropython installed, and a Debian VM also with Micropython installed. For comparison purposes we also conduct tests with a Docker/Micropython container and a Micropython process.

Generally, the results (Figure 14) show that the unikernel's memory usage is fairly close to that of Docker containers. The Tinyx curve is higher, since multiple copies of the Linux kernel are running; however, the additional memory consumption is not dramatic: for 1,000 guests, the system uses

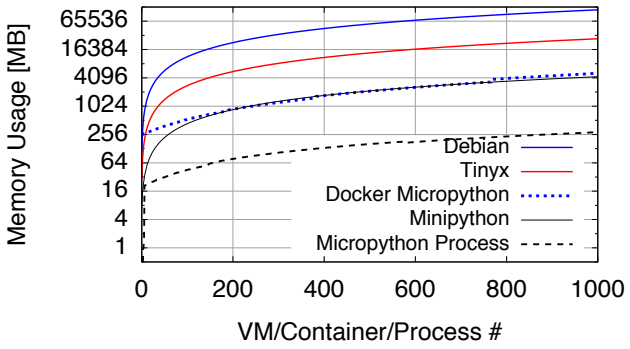


Figure 14: Scalability of VM memory usage for different VMs, for containers and for processes.

about 27GB versus 5GB for Docker. This 22GB difference is small for current server memory capacity (100s of GBs or higher) and memory prices. Debian consumes about 114GB when running 1,000 VMs (assuming 111MB per VM, the minimum needed for them to run).

6.4 CPU Usage

For the final test of the section we take a look at CPU usage when using a `noop` unikernel, Tinyx and a Debian VM, and plot these against usage for Docker. For the VM measurements we use `iostat` to get `Dom0`'s CPU usage and `xentop` to get the guests' utilizations.

As shown in Figure 15, Docker containers have the lowest utilization although the unikernel is only a fraction of a percentage point higher. Tinyx also fares relatively well, reaching a maximum utilization of about 1% when running 1,000 guests. The Debian VM scales more poorly, reaching about 25% for 1,000 VMs: this is because each Debian VM runs a number of services out of the box that, taken together, results in fairly high CPU utilization. In all, the graph shows that CPU utilizations for VMs can be roughly on par with that of containers, as long as the VMs are trimmed down to include only the functionality crucial for the target application.

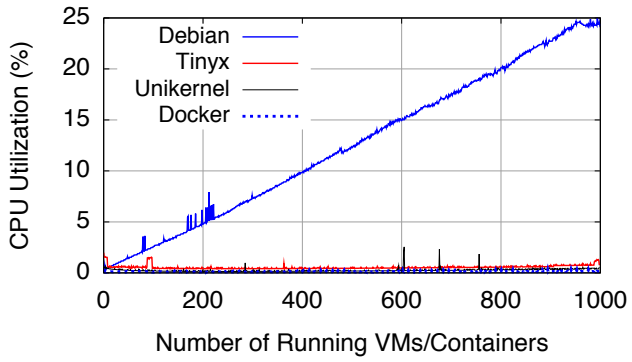


Figure 15: CPU usage for a unikernel, Tinyx, a Debian VM and Docker.

7 USE CASES

We now explore scenarios where lightweight virtualization can bring a tangible benefit over the status quo. In all the following scenarios, using containers would help performance but weaken isolation, while using full-blown VMs would provide the same isolation as lightweight VMs, but with poorer performance.

7.1 Personal Firewalls

The number of attacks targeting mobile phones is increasing constantly [30]. Running up-to-date firewalls and intrusion detection/prevention on the mobile phone is difficult, so one option is to scrub the traffic in the cloud instead. Ideally, each mobile user should be able to run a personal firewall that is on-path of the traffic to avoid latency inflation.

Recently, mobile operators have started working on mobile-edge computing (MEC) [16], which aims to run processing as close as possible to mobile users and, to this end, deploys servers co-located with mobile gateways situated at or near the cellular base stations (or cells).

The MEC is an ideal place to instantiate personal firewalls for mobile users. The difficulty is that the amount of deployed hardware at any single cell is very limited (one or a few machines), while the number of active users in the cell is on the order of a few thousand. Moreover, users enter and leave the cell continuously, so it is critical to be able to instantiate, terminate and migrate personal firewalls quickly and cheaply, following the user through the mobile network.

Using full-blown Linux VMs is not feasible because of their large boot times (a few seconds) and their large image sizes (GBs) which would severely increase migration duration and network utilization. Using containers, on the other hand, is tricky because malicious users could try to subvert the whole MEC machine and would be able to read and tamper with other users' traffic.

Instead, we rely on ClickOS, a unikernel specialized for network processing [29] running a simple firewall configuration we have created. The resulting VM image is 1.7MB in size and the VM needs just 8MB of memory to run; we can run as many as 8000 such firewalls on our 64-core AMD machine, and booting one instance takes about 10ms. Migrating a ClickOS VM over a 1Gbps, 10ms link takes just 150ms.

We want to see if the ClickOS VMs can actually do useful work when so many of them are active simultaneously. We start 1000 VMs running firewalls for 1000 emulated mobile clients, and then run an increasing number of `iperf` instances, each instance representing one client and being serviced by a dedicated VM. We limit each client's throughput to 10Mbps to mimic typical 4G speeds in busy cells. We measure total throughput as well added latency. For the latter, we have one client run ping instead of iperf.

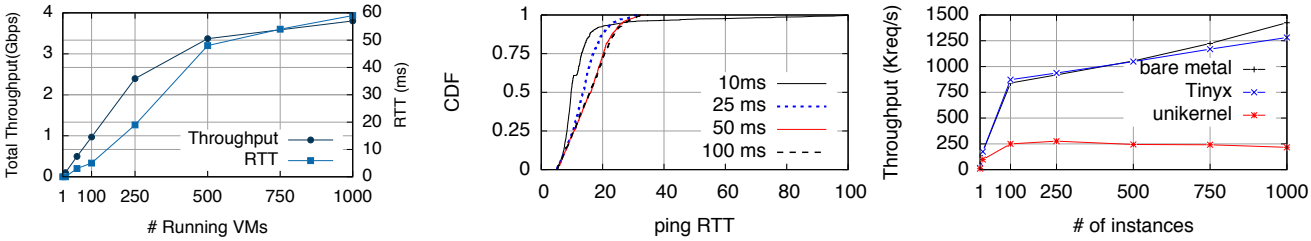
The results, run on a server with an Intel Xeon E5-2690 v4 2.6 GHz processor (14 cores) and 64GB of RAM, are shown in Figure 16a. The cumulative throughput grows linearly until 2.5Gbps (250 clients), each client getting 10Mbps. After that, heavy CPU contention curbs the throughput increase: the average per-user throughput is 6.5Mbps when there are 500 active users, and 4Mbps with 1000 active users. The per-packet added latency is negligible with few active users (tens), but increases to 60ms when 1000 users are active; this is to be expected since the Xen scheduler will effectively round-robin through the VMs. We note that VMs servicing long flows care less about latency, and a better scheduler could prioritize VMs servicing few packets (like our ping VM); this subject is worth future investigation.

To put the performance in perspective, we note that the maximum theoretical download throughput of LTE-advanced is just 3.3Gbps (per cell sector), implying that a single machine running LightVM would be able to run personalized firewalls for all the users in the cell without becoming a performance bottleneck.

7.2 Just-in-Time Service Instantiation

Our second use-case also targets mobile edge computing: we are interested in dynamically offloading work from the mobile to the edge as proposed by [28]; the most important metrics are responsiveness and the ability to offer the service to as many mobile devices as possible. We implemented a dummy service that boots a VM whenever it receives a packet from a new client, and keeps the VM running as long as the client is actively sending packets; after 2s of inactivity we tear down the VM. To measure the worst-case client perceived latency, we have each client send a single ping request, and have the newly booted VM reply to pings.

We use open-loop client arrivals with different intensities and plot the CDFs of ping times in Figure 16b. The different



(a) Running personal firewalls for 1000 users using ClickOS.

(b) Just-in-time instantiation of VMs to service mobile clients.

(c) TLS termination throughput for up to 1000 end points.

Figure 16: LightVM use cases.

curves correspond to varying client inter-arrival rates: with one new client every 25 ms, the client-measured latency is 13ms in the median and 20ms at the 90%. With one new client every 10 ms, the RTTs improve up to the point that our Linux bridge is overloaded and starts dropping packets (mostly ARP packets), hence some pings time out and there is a long tail for the client-perceived latency.

7.3 High Density TLS Termination

The Snowden leaks have revealed the full extent of state-level surveillance, and this has pushed most content providers to switch to TLS (i.e., HTTPS) to the point where 70% of Internet traffic is now encrypted [39]. TLS, however, requires at least one additional round-trip time to setup, increasing page load times significantly if the path RTT is large. The preferred solution to reduce this effect is to terminate TLS as close to the client as possible using a content-distribution network and then serve the content from the local cache or fetch it from the server over a long-term encrypted tunnel.

Even small content providers have started relying on CDNs, which now must serve a larger number of customers on the same geographically-distributed server deployments. TLS termination needs the long term secret key of the content provider, requiring strong isolation between different content-providers' HTTPS proxies; simply running these in containers is unacceptable, while running full Linux VMs is too heavyweight.

We have built two lightweight VMs for TLS termination: one is based on Minipython and the other one is based on Tinyx. Our target is to support as many TLS proxies as possible on a single machine, so we rely on `axtls` [12], a TLS library for embedded systems optimized for size. The unikernel relies on the `lwip` networking stack, boots in 6ms and uses 16MB of RAM at runtime. The Tinyx machine uses 40MB of RAM and boots in 190ms.

To understand whether the CDN provider can efficiently support many simultaneous customers on the same box, we have N `apachebench` clients continuously requesting the same empty file over HTTPS from N virtual machines. We

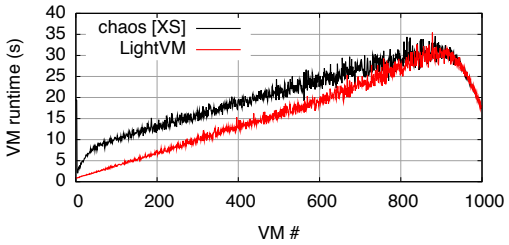


Figure 17: Lightweight compute function service time on an overloaded machine (Minipython unikernels)

measure the aggregate throughput on the 14-core machine and plot it in Figure 16c (bare metal means a Linux process, i.e., no hypervisor). The graph shows that adding more VMs increases total throughput; this is because more VMs fully utilize all CPUs to perform public-key operations, masking protocol overheads. Tinyx's performance is very similar to that of running processes on a bare-metal Linux distribution: around 1400 requests per second are serviced. This number is low because we use 1024-bit RSA keys instead of more efficient variants such as ECDHE. Finally, note that the unikernel only achieves a fifth of the throughput of Tinyx; this is mostly due to the inefficient `lwip` stack. With appropriate engineering, the stack can be fixed, however reaching the maturity of the Linux kernel TCP stack is a tall order.

These results highlight the tradeoffs one must navigate when aiming for lightweight virtualization: either use Tinyx or other minimalistic Linux-based VMs that are inherently slower to boot (hundreds of ms) and more memory hungry, or invest considerable engineering effort to develop minimal unikernels that achieve millisecond-level boot times and allow for massive consolidation.

7.4 Lightweight Compute Service

On-demand compute services such as Amazon's Lambda or blockchain-backed distributed ledgers are increasing in popularity. Such services include data, image, or video aggregations and conversions, or cryptographic hash computations, and perform calculations that often do not last more than a

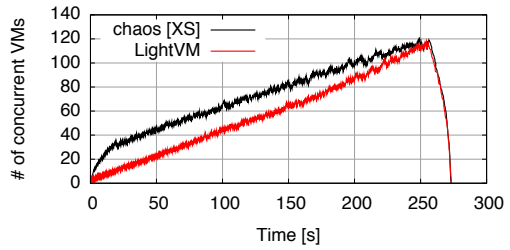


Figure 18: Number of concurrently running Minipython unikernels for the compute service use case.

few seconds. Further, there is no need to keep state between independent calculations which means that the service can be simply destroyed after finishing the calculation. Nevertheless, compute services for different tenants need strong isolation to reduce sensitive information leaks.

Lightweight VMs are a perfect match for such lightweight computation services. For this use case we rely on the Minipython unikernel we created which runs computations written in Python, similar to Amazon's Lambda. The unikernel uses the the lightweight MicroPython interpreter and also links a networking stack. In addition, we have implemented a daemon in `Dom0` that receives compute service requests (in the form of python programs) and spawns a VM to run the program. When the program finishes the VM shuts down. We ran experiments on our four-core machine. All compute services calculated an approximation of e that takes approximately 0.8 seconds. The domains were spawned on three of the four cores (with the fourth exclusively used by `Dom0`).

We generate one thousand compute requests in an open loop with inter-arrival times of 250ms. This is faster than our machine can cope (266ms inter-arrivals lead to full utilization), slowly increasing load on the system. Creation times are not strongly affected by the increasing load, since `Dom0` had its own dedicated core. Creation times for `noxs` slowly increase from approximately 2.8 ms to approximately 3.5 ms. Using the split toolstack and its pre-created domains takes a nearly constant 1.3 ms regardless of the number of already-created domains. Figure 17 shows the time it takes for the nth compute request to be serviced in this overloaded system, and Figure 18 shows the number of active VMs as a function of time. Notice how our optimizations, in particular not using the XenStore, improve the completion times by a factor of 5 when the system is slightly overloaded (100-200 backlogged VMs); here the work reduction provided by `noxs` allows other VMs to do useful work instead, reducing the number of backlogged VMs.

8 RELATED WORK

A number of OS-level virtualization technologies exist and are widely deployed, including Docker, LXC, FreeBSD jails and Linux-VServer, among others [6, 13, 25, 42]. In terms of high density, the work in [17] shows how to run 10,000 Docker containers on a single server. Zhang et al. implement network functions using Docker containers and can boot up to 80K of them [51]. In our work, we make the case for lightweight virtualization on top of a hypervisor, providing strong isolation while retaining the attractive properties commonly found in containers.

A number of works have looked into optimizing hypervisors such as Xen, KVM and VMWare [3, 20, 47] to reduce their overheads in terms of boot times and other metrics. For example, Intel Clear Containers [45] (ICC) has similarities to our work but a different goal. ICC tries to run containers within VMs with the explicit aim of keeping compatibility with existing frameworks (Docker, rkt); this compatibility results in overheads. LightVM optimizes both the virtualization system and guests, achieving performance similar to or better than containers without sacrificing isolation. We have also showed that it is possible to make automated build tools to minimize the effort needed to make such specialized guests (i.e., with Tinyx). ICC also optimizes parts of the virtualization system (e.g., the toolstack), but does not provide other features such as the split toolstack and uses larger guests: an ICC guest is 70MB and boots in 500ms [19] as opposed to a Tinyx one which is about 10MB and boots in about 300ms.

`ukvm` [50] implements a specialized unikernel monitor on top of KVM and uses MirageOS unikernels to achieve 10 ms boot times (the main metric the work focuses on). Jitsu [26] optimizes parts of Xen to implement just-in-time instantiation of network services by accelerating connection start-up times. Earlier work [48] implemented JIT instantiation of honeypots through the use of image cloning; unlike the work there, we do not require the VMs on the system to run the same application in order to achieve scalability. The work in [36] optimizes xl toolstack overheads, but their use of Linux VMs results in boot times in the hundreds of milliseconds or seconds range. LightVM aims to provide container-like dynamics; as far as we know, this is the first proposal to simultaneously provide small boot, suspend/resume and migration times (sometimes an order of magnitude smaller than previous work), high density and low per-VM memory footprints.

Beyond containers and virtual machines, other works have proposed the use of minimalistic kernels or hypervisors to provide lightweight virtualization. Exokernel [8] is a minimalistic operating system kernel that provides applications with the ability to directly manage physical resources. NOVA [44] is a microhypervisor consisting of a thin virtualization layer and thus aimed at reducing the attack surface (NOVA's TCB is about 36K LoC, compared to for instance 11.4K for Xen's ARM port). The Denali isolation kernel is able to boot 10K VMs but does not support legacy OSes and

has limited device support [49]. The work in [40] proposes the implementation of cloudlets to quickly offload services from mobile devices to virtual machines running in a cluster or data center, although the paper reports VM boot times in the 60-90 seconds range. The Embassies project [7, 15] and Drawbridge [33] make a similar case than us by providing strong isolation through picoprocesses that have a narrow interface (the way VMs do), though the target there was running isolated user-space applications or web client apps.

Finally, unikernels [27] have seen significant research interest; prior work includes Mirage [27], ClickOS [29], Erlang on Xen [9] and OSv [21] to name just a few. Our work does not focus on unikernels, but rather leverages them to be able to separate the effects coming from the virtual machine from those of the underlying virtualization platform. For example, they allow us to reach high density numbers without having to resort to overly expensive servers.

9 DISCUSSION AND OPEN ISSUES

Memory sharing: LightVM does not use page sharing between VMs, assuming the worst-case scenario where all pages are different. One avenue of optimization is to use memory de-duplication (as proposed by SnowFlock [24]) to reduce the overall memory footprint; unfortunately this requires non-negligible changes to the virtualization system.
Generality: While LightVM is based on Xen, most of its components can be extended to other virtualization platforms such as KVM. This includes (1) the optimized toolstack, where work such as `ukvm` [50] provides a lean toolstack for KVM (among other things); (2) the pre-creation of guests, which is independent of the underlying hypervisor technology; and (3) the use of specialized OSes and unikernels, several of which already exist for non-Xen hypervisors (e.g., the Solo5 unikernel [18], rump kernels [38], OSv [21]). The one feature that is Xen-specific is the XenStore, though KVM keeps similar information in the Linux kernel (process information) and in the QEMU process (device information).
Usability and portability: Despite its compelling performance, LightVM is still not as easy to use as containers. Container users can rely on a large ecosystem of tools and support to run unmodified existing applications.

LightVM exposes a clear trade-off between performance and portability/usability. Unikernels provide the best performance, but require non-negligible development time and manual tweaking to get an image to compile against a target application. Further, debugging and extracting the best performance out of them is not always trivial since they do not come with the rich set of tools that OSes such as Linux have. At the other extreme, VMs based on general-purpose OSes such as Linux require no porting and can make use of existing management tools, but their large memory footprints and high boot times, among other issues, have at least partly resulted in the widespread adoption of containers (and their security problems).

In designing and implementing the Tinyx build system we tried to take a first step towards solving the problem: Tinyx provides better performance than a standard Debian distribution without requiring any application porting. However, Tinyx is still a compromise: we sacrifice some performance with respect to unikernels in order to keep the ecosystem and existing application support.

The ultimate goal is to be able to automatically build custom-OSes targeting a single application. For instance, the Rump Kernels project [38] builds "unikernels", relying on large portions of NetBSD to support existing applications. This is not quite what we would need since the performance and size of the resulting images are not in the same order of magnitude as LightVM. Part of the solution would have to decompose an existing OS into fine-granularity modules, automatically analyze an application's dependencies, and choose which is the minimum set of modules needed for the unikernel to compile. This area of research is future work.

10 CONCLUSIONS

We have presented LightVM, a complete redesign of Xen's toolstack optimized for performance that can boot a minimalistic VM in as little as 2.3ms, comparable to the `fork`/`exec` implementation in Linux (1ms). Moreover, LightVM has almost constant creation and boot times regardless of the number of running VMs; this is in contrast to the current toolstack that can take as much as 1s to create a VM when the system is loaded. To achieve such performance LightVM foregoes Xen's centralized toolstack architecture based on the XenStore in favor of a distributed implementation we call `noxs`, along with a reimplementation of the toolstack.

The use cases we presented show that there is a real need for lightweight virtualization, and that it is possible to simultaneously achieve both good isolation and performance on par or better than containers. However, there is a development price to be paid: unikernels offer best performance but require significant engineering effort which is useful for highly popular apps (such as TLS termination) but likely too much for many other applications. Instead, we have proposed Tinyx as a midway point: creating Tinyx images is streamlined and (almost) as simple as creating containers, and performance is on par with that of Docker containers.

ACKNOWLEDGMENTS

This paper has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement no. 671566 ("Superfluidity").

REFERENCES

[1] Amazon Web Services [n. d.]. Amazon EC2 Container Service. https://aws.amazon.com/ecs/. ([n. d.]).

[2] Amazon Web Services [n. d.]. AWS Lambda - Serverless Compute. https://aws.amazon.com/lambda. ([n. d.]).

[3] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. 2003. Xen and the Art of Virtualization. *SIGOPS Oper. Syst. Rev.* 37, 5 (Oct. 2003), 164–177. https://doi.org/10.1145/1165389.945462

[4] J. Clark. [n. d.]. Google: "EVERYTHING at Google runs in a container". http://www.theregister.co.uk/2014/05/23/google_containerization_two_billion/. ([n. d.]).

[5] Patrick Colp, Mihir Nanavati, Jun Zhu, William Aiello, George Coker, Tim Deegan, Peter Loscocco, and Andrew Warfield. 2011. Breaking Up is Hard to Do: Security and Functionality in a Commodity Hypervisor. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP '11)*. ACM, New York, NY, USA, 189–202. https://doi.org/10.1145/2043556.2043575

[6] Docker [n. d.]. The Docker Containerization Platform. https://www.docker.com/. ([n. d.]).

[7] John R. Douceur, Jeremy Elson, Jon Howell, and Jacob R. Lorch. 2008. Leveraging Legacy Code to Deploy Desktop Applications on the Web. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*. USENIX Association, Berkeley, CA, USA, 339–354. http://dl.acm.org/citation.cfm?id=1855741.1855765

[8] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. 1995. Exokernel: An Operating System Architecture for Application-level Resource Management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles (SOSP '95)*. ACM, New York, NY, USA, 251–266. https://doi.org/10.1145/224056.224076

[9] Erlang on Xen 2012. Erlang on Xen. http://erlangonxen.org/. (July 2012).

[10] Google Cloud Platform [n. d.]. The Google Cloud Platform Container Engine. https://cloud.google.com/container-engine. ([n. d.]).

[11] A. Grattafiori. [n. d.]. Understanding and Hardening Linux Containers. https://www.nccgroup.trust/us/our-research/understanding-and-hardening-linux-containers/. ([n. d.]).

[12] Cameron Hamilton-Rich. [n. d.]. axTLS Embedded SSL. http://axtls.sourceforge.net. ([n. d.]).

[13] Poul henning Kamp and Robert N. M. Watson. 2000. Jails: Confining the omnipotent root. In *In Proc. 2nd Intl. SANE Conference*.

[14] J. Hertz. [n. d.]. Abusing Privileged and Unprivileged Linux Containers. https://www.nccgroup.trust/uk/our-research/abusing-privileged-and-unprivileged-linux-containers/. ([n. d.]).

[15] Jon Howell, Bryan Parno, and John R. Douceur. 2013. Embassies: Radically Refactoring the Web. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. USENIX, Lombard, IL, 529–545. https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/howell

[16] Yun Chao Hu, Milan Patel, Dario Sabella, Nurit Sprecher, and Valerie Young. 2015. Mobile Edge Computing - A key technology towards 5G. *ETSI White Paper No. 11, First edition* (2015).

[17] IBM. [n. d.]. Docker at insane scale on IBM Power Systems. https://www.ibm.com/blogs/bluemix/2015/11/docker-insane-scale-on-ibm-power-systems. ([n. d.]).

[18] IBM developerWorks Open [n. d.]. Solo5 Unikernel. https://developer.ibm.com/open/openprojects/solo5-unikernel/. ([n. d.]).

[19] Intel. [n. d.]. Intel Clear Containers: A Breakthrough Combination of Speed and Workload Isolation. https://clearlinux.org/sites/default/files/vmscontainers_wp_v5.pdf. ([n. d.]).

[20] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. 2007. KVM: the Linux Virtual Machine Monitor. In *In Proc. 2007 Ottawa Linux Symposium (OLS '07)*.

[21] Avi Kivity, Dor Laor, Glauber Costa, Pekka Enberg, Nadav Har'El, Don Marti, and Vlad Zolotarov. 2014. OSv—Optimizing the Operating System for Virtual Machines. In *Proceedings of the 2014 USENIX Annual Technical Conference (USENIX ATC '14)*. USENIX Association, Philadelphia, PA, 61–72. https://www.usenix.org/conference/atc14/technical-sessions/presentation/kivity

[22] E. Kovacs. [n. d.]. Docker Fixes Vulnerabilities, Shares Plans For Making Platform Safer. http://www.securityweek.com/docker-fixes-vulnerabilities-shares-plans-making-platform-safer. ([n. d.]).

[23] Simon Kuenzer, Anton Ivanov, Filipe Manco, Jose Mendes, Yuri Volchkov, Florian Schmidt, Kenichi Yasukata, Michio Honda, and Felipe Huici. 2017. Unikernels Everywhere: The Case for Elastic CDNs. In *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '17)*. ACM, New York, NY, USA, 15–29. https://doi.org/10.1145/3050748.3050757

[24] Horacio Andrés Lagar-Cavilla, Joseph Andrew Whitney, Adin Matthew Scannell, Philip Patchin, Stephen M. Rumble, Eyal de Lara, Michael Brudno, and Mahadev Satyanarayanan. 2009. SnowFlock: Rapid Virtual Machine Cloning for Cloud Computing. In *Proceedings of the 4th ACM European Conference on Computer Systems (EuroSys '09)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/1519065.1519067

[25] LinuxContainers.org [n. d.]. LinuxContainers.org. https://linuxcontainers.org. ([n. d.]).

[26] Anil Madhavapeddy, Thomas Leonard, Magnus Skjegstad, Thomas Gazagnaire, David Sheets, Dave Scott, Richard Mortier, Amir Chaudhry, Balraj Singh, Jon Ludlam, Jon Crowcroft, and Ian Leslie. 2015. Jitsu: Just-In-Time Summoning of Unikernels. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI '15)*. USENIX Association, Oakland, CA, 559–573. https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/madhavapeddy

[27] Anil Madhavapeddy and David J. Scott. 2013. Unikernels: Rise of the Virtual Library Operating System. *Queue* 11, 11, Article 30 (Dec. 2013), 15 pages. https://doi.org/10.1145/2557963.2566628

[28] Y. Mao, J. Zhang, and K. B. Letaief. 2016. Dynamic Computation Offloading for Mobile-Edge Computing With Energy Harvesting Devices. *IEEE Journal on Selected Areas in Communications* 34, 12 (Dec 2016), 3590–3605. https://doi.org/10.1109/JSAC.2016.2611964

[29] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. 2014. ClickOS and the Art of Network Function Virtualization. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI '14)*. USENIX Association, Seattle, WA, 459–473. https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/martins

[30] McAffee. 2016. Mobile Threat Report. https://www.mcafee.com/us/resources/reports/rp-mobile-threat-report-2016.pdf. (2016).

[31] MicroPython [n. d.]. MicroPython. https://micropython.org/. ([n. d.]).

[32] Microsoft. [n. d.]. Azure Container Service. https://azure.microsoft.com/en-us/services/container-service/. ([n. d.]).

[33] Microsoft Research. [n. d.]. Drawbridge. https://www.microsoft.com/en-us/research/project/drawbridge/. ([n. d.]).

[34] minios [n. d.]. Mini-OS. https://wiki.xenproject.org/wiki/Mini-OS. ([n. d.]).

[35] A. Mourat. [n. d.]. 5 security concerns when using Docker. https://www.oreilly.com/ideas/five-security-concerns-when-using-docker. ([n. d.]).

[36] Vlad Nitu, Pierre Olivier, Alain Tchana, Daniel Chiba, Antonio Barbalace, Daniel Hagimont, and Binoy Ravindran. 2017. Swift Birth and Quick Death: Enabling Fast Parallel Guest Boot and Destruction in the

Xen Hypervisor. In *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '17)*. ACM, New York, NY, USA, 1–14. https://doi.org/10.1145/3050748.3050758

[37] MAN page. [n. d.]. Linux system calls list. http://man7.org/linux/man-pages/man2/syscalls.2.html. ([n. d.]).

[38] Rumpkernel.org [n. d.]. Rump Kernels. http://rumpkernel.org/. ([n. d.]).

[39] Sandvine. [n. d.]. Internet traffic encryption. https://www.sandvine.com/trends/encryption.html. ([n. d.]).

[40] Mahadev Satyanarayanan, Paramvir Bahl, Ramón Caceres, and Nigel Davies. 2009. The Case for VM-Based Cloudlets in Mobile Computing. *IEEE Pervasive Computing* 8, 4 (Oct. 2009), 14–23. https://doi.org/10.1109/MPRV.2009.82

[41] Justine Sherry, Shaddi Hasan, Colin Scott, Arvind Krishnamurthy, Sylvia Ratnasamy, and Vyas Sekar. 2012. Making Middleboxes Someone Else's Problem: Network Processing As a Cloud Service. In *Proceedings of the ACM SIGCOMM 2012 Conference on Computer Communication (SIGCOMM '12)*. ACM, New York, NY, USA, 13–24. https://doi.org/10.1145/2342356.2342359

[42] Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson. 2007. Container-based Operating System Virtualization: A Scalable, High-performance Alternative to Hypervisors. *SIGOPS Oper. Syst. Rev.* 41, 3 (March 2007), 275–287. https://doi.org/10.1145/1272998.1273025

[43] S. Stabellini. [n. d.]. Xen on ARM. http://www.slideshare.net/xen_com_mgr/alsf13-stabellini. ([n. d.]).

[44] Udo Steinberg and Bernhard Kauer. 2010. NOVA: A Microhypervisor-based Secure Virtualization Architecture. In *Proceedings of the 5th European Conference on Computer Systems (EuroSys '10)*. ACM, New York, NY, USA, 209–222. https://doi.org/10.1145/1755913.1755935

[45] A. van de Ven. [n. d.]. An introduction to Clear Containers. https://lwn.net/Articles/644675/. ([n. d.]).

[46] Akshat Verma, Gargi Dasgupta, Tapan Kumar Nayak, Pradipta De, and Ravi Kothari. 2009. Server Workload Analysis for Power Minimization Using Consolidation. In *Proceedings of the 2009 USENIX Annual Technical Conference (USENIX ATC '09)*. USENIX Association, Berkeley, CA, USA, 28–28. http://dl.acm.org/citation.cfm?id=1855807.1855835

[47] VMWare. [n. d.]. vSphere ESXi Bare-Metal Hypervisor. http://www.vmware.com/products/esxi-and-esx.html. ([n. d.]).

[48] Michael Vrable, Justin Ma, Jay Chen, David Moore, Erik Vandekieft, Alex C. Snoeren, Geoffrey M. Voelker, and Stefan Savage. 2005. Scalability, Fidelity, and Containment in the Potemkin Virtual Honeyfarm. *SIGOPS Oper. Syst. Rev.* 39, 5 (Oct. 2005), 148–162. https://doi.org/10.1145/1095809.1095825

[49] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. 2002. Scale and Performance in the Denali Isolation Kernel. *SIGOPS Oper. Syst. Rev.* 36, SI (Dec. 2002), 195–209. https://doi.org/10.1145/844128.844147

[50] Dan Williams and Ricardo Koller. 2016. Unikernel Monitors: Extending Minimalism Outside of the Box. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud '16)*. USENIX Association, Denver, CO. https://www.usenix.org/conference/hotcloud16/workshop-program/presentation/williams

[51] Wei Zhang, Jinho Hwang, Shriram Rajagopalan, K.K. Ramakrishnan, and Timothy Wood. 2016. Flurries: Countless Fine-Grained NFs for Flexible Per-Flow Customization. In *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies (CoNEXT '16)*. ACM, New York, NY, USA, 3–17. https://doi.org/10.1145/2999572.2999602