CS-477: Virtualization

Lecture Notes

November 26, 2024

Contents

What is virtualization?

Virtualization is about wanting to create a logical view of the physical resources that exist. We virtualize resources: CPU, memory, I/O. We partition the resources in a logical way instead of the physical reality. And this is mostly used for portability and flexibility.

For instance, let's say we have a whole machine with its resources. We want to mimick the behavior of the physical entity.

Let's compare a traditional operating system with virtual machines. In a traditional OS, the kernel runs along the processes on the physical hardware. Instead, we have two forms of virtual machines: either we have a kernel with another kernel running inside, or we have a hypervisor that runs two or more virtual machines. Each VM has its own kernel and apps.

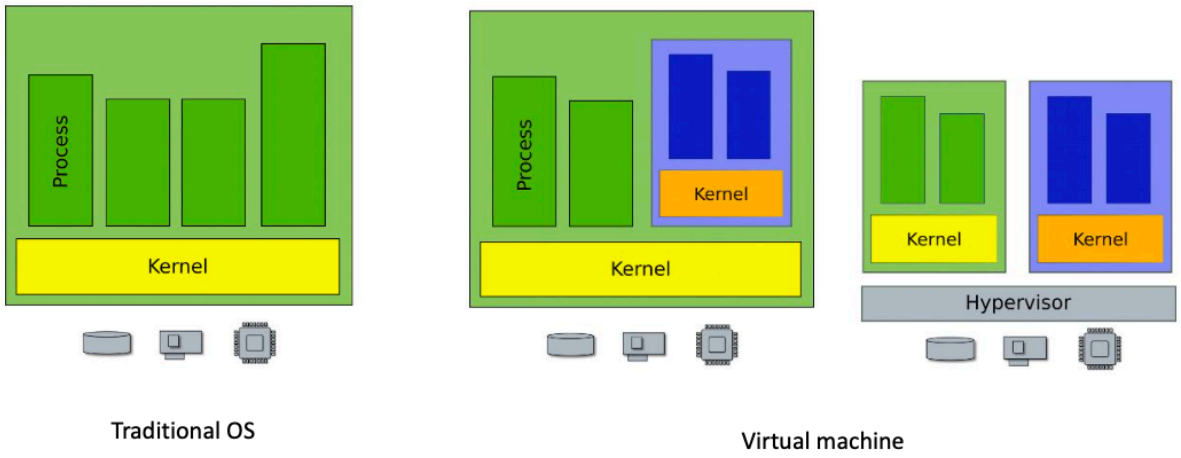

Figure 1: Comparison between a traditional OS and a virtual machine.

A **virtual machine** is an execution environment identical to a physical machine, with the ability to execute a full OS. There is a virtualization layer called a virtual machine monitor (VMM) or hypervisor. It is to an operating system what an operating system is to a process. Initially, when these terms were proposed, there was a slight difference between the two, but today it is not really treated like that: a virtual machine monitor is the same thing as an hypervisor, there is no difference today.

History

VM research started in the 60-70s. Back in the days, they had a very different reason: they used mainframes systems that were bigger than this room and multiple people working on it. We want people to be able to run multiple users one at the time and multiplex them. This was the case for virtualization. In the 80s-90s, with smaller machines people lost the interest in virtualization. Operating systems were already multi-user, and we weren't using machines to the extent of today. Now we have data centers, and want horizontal scaling of resources, so companies and industries wanted to use it for their infrastructure. This is when virtualization came back. It catered to different markets at the time and it is still happening today.

Types

With virtualizations, we can run OSes and user apps on the same hardware. There are two ways to achieve virtualization:

- **Type-1**: The hypervisor actually runs on the hardware. For instance, Xen, VMWare ESX, Microsoft Hyper-V. The *host* is the virtual machine monitor and the hardware itself.
- **Type-2**: the hypervisor is working with the host OS or in conjunction with. Your VMM and OS are working together, not completely on top of it. For instance: KVM, VMWare Workstation, Sun VirtualBox, QEMU.

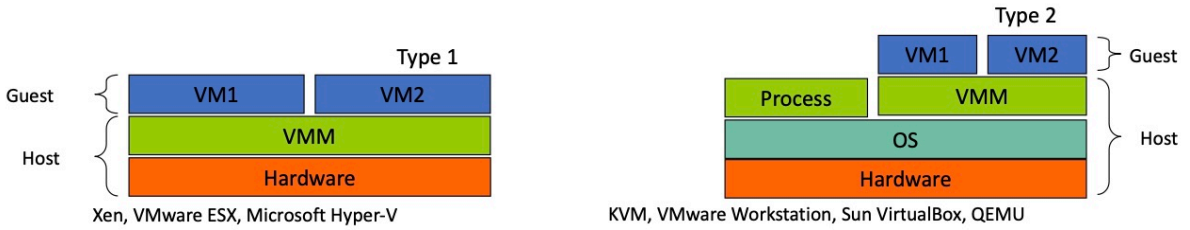

Figure 2: Types of virtualization

Xen was the one that was used a lot in the early 2000s. KVM came from a startup.

In 2009 they merged this whole project to the Linux mainline, and became what is known as KVM today. As of now, KVM is the most widely used hypervisor by almost all of the hyperscalers except Microsoft and VMWare. Google, Oracle and Amazon all use KVM or a sort of it. It's used a lot because of the support it has. Xen is not used as much except for a few companies.

Advantages

Virtualization provides many advantages:

- **Isolation**: we want to provide isolation for security.
- **Rollback**: virtual machines make it easier to rollback environments. Since they are logical representations, we don't have to think about the underlying hardware.
- **Abstraction**: since you're abstracting hardware out, you can run a virtual machine any way you want, you don't need to rely on a specific sort of hardware or OS. For instance, if you boot a machine and want to run images, they would not necessarily work on the hardware itself. But with virtualization, you can come with a minimal representation that you can run across hypervisors and devices. You can do this experiment when you can run a simple VM and run `lsmod` to see the set of devices that have been set up. `lsmod` is the set of kernel modules that are running, so it is an indirect representation of the devices present in your system. The other way would be to use `lspci`, which will tell you about all the PCI devices that are attached to the machine.
- **Portability**: with virtual machines you can easily switch to different VMs. Since you are not specializing for a special hardware, you can distribute the workload and run the VM anywhere. For instance today, you can run an app one any of the datacenters in any region, it doesn't matter, and it runs out of the box. It might not be ultra-performant but it will work.
- **Deployment**: it gives you flexibility. You can have very fast development cycles.

It is a big advantage when you use VMs and virtualization.

Applications

There are a lot of applications that have been proposed. For example, you might have seen:

- **Server virtualization** is something which you see every time you talk about the Xen VM, all these kind of things.
- **Desktop virtualization**: you are just virtualizing the desktop (for instance VirtualBox or Citrix).
- **Mobile virtualization**: you can have two different operating systems running on the same hardware, even on mobile devices.
- **Cloud computing** is a category with many examples:
  - **IaaS** (Infrastructure as a Service): you provide the whole VM to run.
  - **PaaS** (Platform as a Service), also called serverless.
  - **SaaS** (Software as a Service), you run your whole product, like for instance Adobe Createive Cloud. The app can be virtualized and access online, you don't even need to install anything on your machines. Examples include Microsoft Office 365 and Google Drive.
- **Emulation** can also be useful.

Virtualizing techniques

How can you implement virtualization?

Trap and emulate

The classic example is to use the **trap and emulate model**. When you design a hypervisor you think in instructions.

There are instructions that are privileged and non-privileged. Privileged have to be executed by the ring 0 and not in ring 3. There are also sensitive instructions, which change the processor and modify the hardware. Their behavior is different in user and kernel mode. But you might not trap if you execute it. It's possible to do them in userspace and it's a huge problem with the classical approach to virtualization, where you have to annotate these instructions and annotate them.

It's similar to the link between process and OS. If the guest OS wants to change something about the hardware, it has to be handled by the hypervisor and trapped/emulated.

One way is **paravirtualization**. The guest OS knows it's running as guest. Instead of calling the privileged instruction `cli`, it does a **hypercall**, which is a syscall. The hypervisor then does the operation on behalf of the guest OS. With paravirtualization, you can get near-native performance and you don't need hardware support, but you need to modify all the apps. Xen uses paravirtualization, and KVM also provides a paravirtualization interface which is used when the hardware does not support some resources.

Full virtualization

There are three ways of implementing full virtualization:

- **Emulation**: In this case we're emulating the whole CPU in software.
- **Binary translation**: we do some sort of just-in-time translation. The problem is that it's hard to implement. It's faster than emulation but still slow.
- **Hardware-assisted virtualization**: the hardware itself provides ways to implement virtualization. What they do is that they define a different mode (VM/VMM mode). When the VM runs it does a **VM entry operation**. It can still run in hardware ring 0, but in a separate mode. You can have all the functionality the

guest OS should have. You can execute whenever you want with a VM exit. The hardware does everything for you. By using this, you remove the restrictions of paravirtualization and binary translation and it's still extremely fast.

- For instance, **Intel VMX (VM extension)** supports root and non-root modes. Root is for the host OS and non-root for the guest operating system. It's very similar to transfers from process to OS, but now from guest OS to hypervisor. There is a special VM control structure maintained and updated by the hypervisor to keep track for each CPU of the operations.

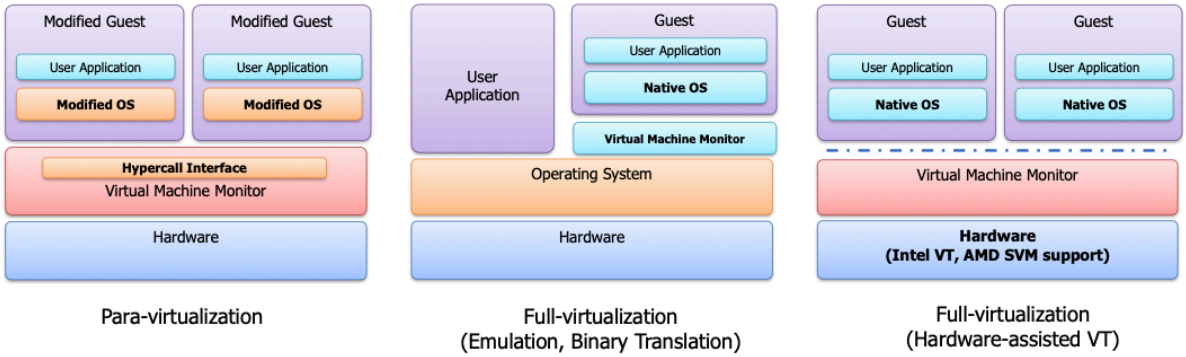

Figure 3: Comparison of virtualization techniques

Comparison

	Para-virtualization	Full-virtualization (Emulation)	Full-virtualization (Binary translation)	Full-virtualization (Hardware-assisted VT)
Speed	Very Fast (Almost Native)	Very Slow	Fast	Fast
Guest Kernel Modification	Yes	No	No	No
Support Other Arch	No	Yes	No	No
Solutions	Xen, VMware ESX	Bochs	VMware, QEMU	KVM, Xen
Purposes	Server virtualization	Emulator	Desktop virtualization	Desktop virtualization

Table 1: Comparison of Virtualization Types

Virtual memory mapping

How can memory get virtualized in a VM? The hypervisor has to give the illusion to the guest OS that it is working with the physical address space, since operating systems

work with physical addresses.

Let's say the hypervisor allocates some memory for the VM. It will allocate a block of *host virtual addresses* and expose it to the VM. When the VM gets it, it thinks that it has gotten a physical address. But it is not really a physical address, it is just a guest physical address. And meanwhile, applications running on the VM have their own layer, which is the guest virtual addresses.

When applications access something, the CPU will translate it to the guest physical address, which is what the guest OS is going to work with.

Now if the page that you want to play with is not available, it has to ask the host to get it. There are multiple ways to implement this.

Direct paging

In this approach, the guest OS maintains itself a mapping between its virtual addresses (guest virtual addresses) and the physical addresses of the host (host physical addresses). Whenever a logical access is happening, the hardware walks the page tables to determine the host physical addresses to use.

The main issue with this approach is that you need to use a guest kernel that is aware of the host physical memory, which means that you cannot use unmodified operating systems. However, the performance is excellent, since there is no overhead induced by virtualization. This approach isn't used frequently.

Shadow paging

This approach consists of the virtual machine monitor monitoring the guests's virtual to physical address mapping, and creating shadow page tables that it exposes to the hardware. This causes some overhead because there are two levels of page tables being used (the one created by the guest OS and the shadow page table created by the VMM). The VMM marks the guest page tables as read-only, so that it can trap whenever they change and synchronize the shadow page tables. However, this ap-

proach is commonly used because it does not require support from the guest OS and hardware.

Nested paging

The VMM can maintain a mapping ("nested page tables") from the guest pyhsical addresses to the host physical addresses, which is then handled by the hardware itself. The hardware sees both the guest page tables and the nested page tables: when a logical address is accessed by the VM, the hardware will walk both the guest page table and then the nested page table to determine the host physical address to use.

This approach is simple to implement and supports all guest operating systems, but it requires hardware support and takes more space in the TLB since information about both page tables is needed.

Extended page tables (EPT) is the implementation used by some CPUs made by Intel. The hypervisor has an *EPT base pointer* in the VM control structure that points to the extended page tables. They are used along with the guest page tables to resolve addresses when the VM is running. The guest OS has full control over its page tables, the related control registers (CR0, CR3, CR4 paging bits), and events such as page faults and TLB invalidation. The EPT entries also specify the privilege level that the software has when accessing the addresses (disallowed accesses are called *EPT violations* and cause VM exits).

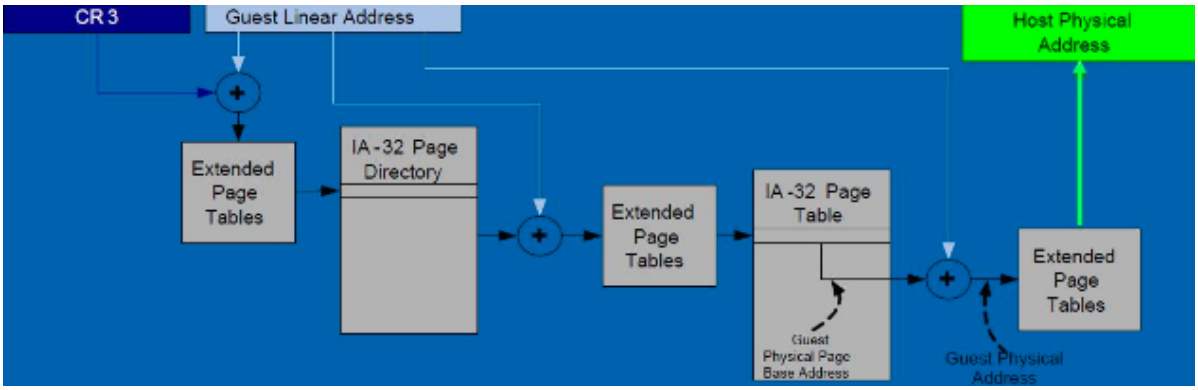

Figure 4: Address translation using extended page tables

Comparison

	Direct Paging	Shadow Paging	Nested Paging
Speed	Very Fast (Almost Native)	Very Slow	Fast
Guest Kernel Modification	Yes	No	No
Need H/W Support	No	No	Yes
Complexity	Simple	Complex	Very Simple

Table 2: Comparison of implementations of memory mapping virtualization

Of course, nested paging is the best when the hardware supports it. Otherwise, direct paging is the fastest but necessitates changes in the guest kernel, which are not required when using shadow paging.

I/O virtualization

How do we handle I/O interrupts? There are three models to handle them.

Front-end/Back-end driver model

The guest OS uses a paravirtualized frontend driver, which talks to the backend driver. The backend driver can be Dom0 (a privileged Xen domain) and the front-end driver is DomU (the same domain as the one the VM itself is running on). The backend driver serves as a server who receives all incoming requests and applies them to the actual hardware.

This model is used by a lot of hypervisors, such as Xen and basic usages of qemu.

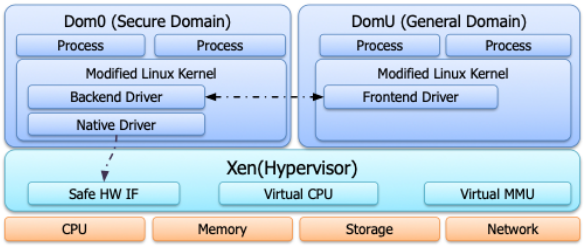

Figure 5: The front-end/back-end driver model in Xen

Emulation

The entire device is emulated in software itself. The guest OS accesses the device using the regular device driver, and the VMM will intercept device interrupts. It's used a lot in qemu.

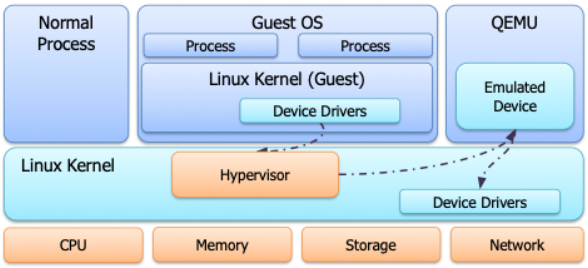

Figure 6: I/O virtualization by emulation

Hardware-assisted I/O virtualization

The hardware used on the machine can actually support requests coming from multiple guest OSes, who use the regular device drivers.

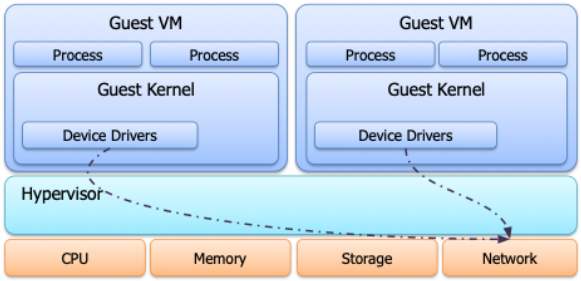

Figure 7: Hardware-assisted I/O virtualization

Comparison

	Front-end/Back-end Driver Model	**Emulation**	**H/W Assisted IO Virtualization**
Speed	Very Slow	Very Slow	Fast
Device Driver Modification	Yes	No	No
Need H/W Support	No	No	Yes
Complexity	Simple	Complex	Simple

Paper: *My VM is Lighter (and Safer) than your Container*

This week's paper is the easiest paper to understand and read. It took me 25 minutes to read.

When you want to virtualize workloads, you have two models: virtualization or containers.

Containers work by compiling an application, and having a framework with namespaces, which is the basic abstraction used to give the illusion to the app it's being ran separately.

But there are issues with containers:

- They must handle the interaction at the difficult syscall ABI instead of the simpler APIs from x86

- There are many exploits that could exist
- They can exhaust resources

The main question being addressed by this paper is: can we get the performance of containers with the security of VMs? In order to achieve this, the solution would need to be lightweight, provide fast instantiation, run hundreds of instances at the same time, and allow them to be paused/unpaused rapidly.

Reducing the VM size

One key observation is that the boot time increases when we increase the VM image size. For instance, you have to interact with the VMM to initialize resources, which causes many interrupts. And by increasing the VM image size, you have to allocate more stuff.

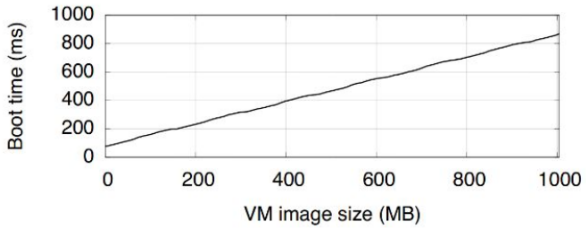

Figure 8: The size of the guest VM image increase the boot time.

But most of the VM is not used! So we would like to minimize the size of the guest OS. The paper proposes ways to have small OS images:

- A **unikernel**, which is a VM that contains a single app with a minimalistic OS implementation. The OS is directly linked to the application.
- **Tinyx** is a Linux distribution that only includes the minimal rqeuirements for the app by finding its dependencies and cleaning up all the rest.

Improving the performance of Xen

The paper also explores ways of improving the performance of Xen.

The important observation that the paper makes is that as the number of VM increases, the creation time of a new VM becomes dominated by the device creation and XenStore. XenStore is responsible for creating the virtual machines, which includes creating devices and updating data structures about the VM. The time taken by XenStore is increasing more than linearly, because there are a lot of parts of XenStore that don't scale well (e.g. reading a linked list each time a unique name must be generated). The entire stack is also quite inefficient, creating a lot of XenStore entries for each device and accessing it frequently.

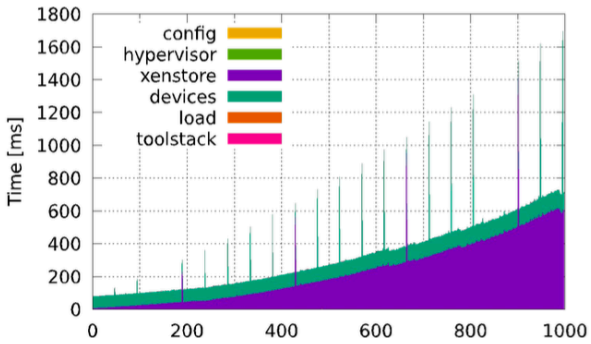

Figure 9: Breakdown of the creation time of a single VM, when the number of created VMs increases.

To improve the situation, the authors of the paper designed **LightVM**, a redesign of the Xen control plane where Xenstore is bypassed for VM creation and boot using a lean driver called **NoXS** (no XenStore).

They also use **libchaos**, a tool they created to improve the creation time of VMs. The idea is that when VMs are created, a lot of operations are common to all virtual machines, and could be made quicker if they were done in advance. It does so by running a daemon that pre-creates optimistically a template of the future virtual machines, that can then be used when the machine is actually needed. During the run-time phase, only the operations that need to be made on top of the common operations are done, which saves time.

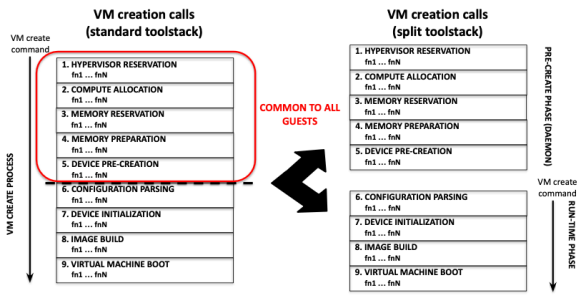

Figure 10: The split toolstack, including th pre-create phase and the run-time phasee.

Evaluation

Creating VMs using unikernels gives much better performance and scales better than Docker. Using Tinyx gives times that are close to the instantiation times of Docker containers.

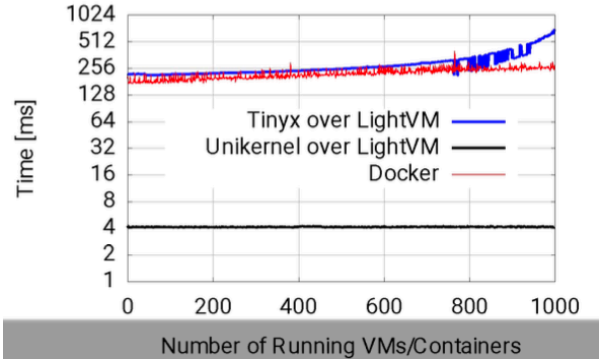

Figure 11: VM instantiation

When creating a very high number of VMs (using a no-operation unikernel), the unikernel performs quicker than Docker, while Docker's times increase with the number of VMs and runs out of memory quickly.

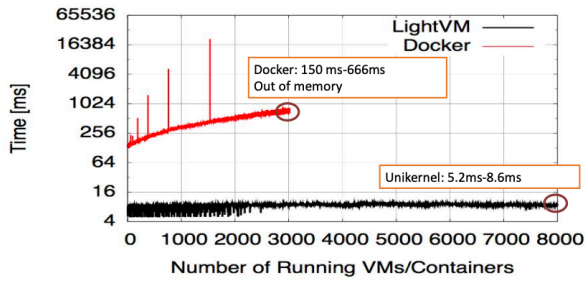

Figure 12: Instantiation in high density

The techniques in this paper also provide good performance and scalability for saving a checkpoint of the VM state and performing it, as well as migrating VMs.

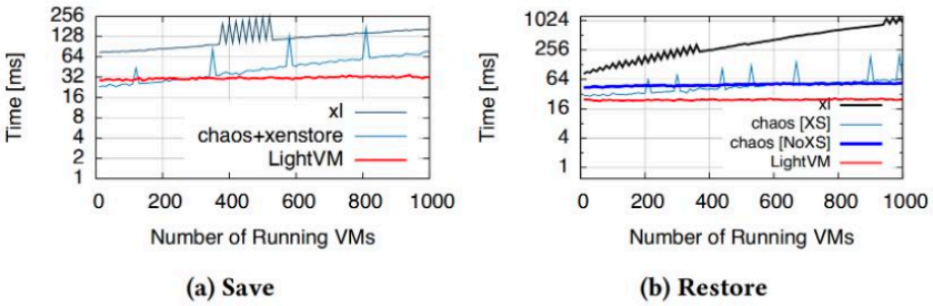

Figure 13: Checkpointing

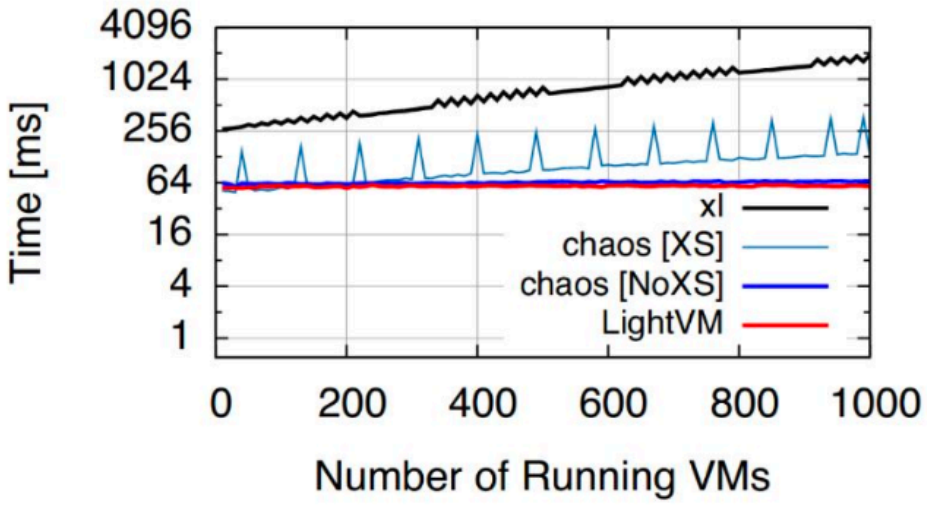

Figure 14: Migration

The memory footprint using the VM creation techniques discussed in the paper is lower than using a regular Debian image, and the CPU usage is comparable to Docker and scales well.

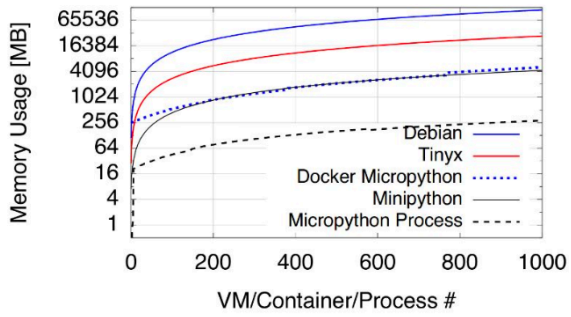

Figure 15: Memory footprint

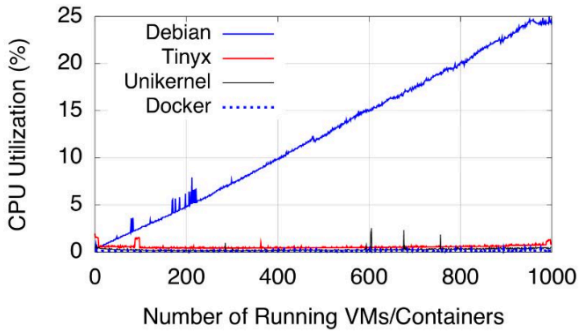

Figure 16: CPU usage