HyperFork: Improving Serverless Latency and Throughput through Virtual Machine Flash-Cloning

Michael Colavita

Advisor: Eddie Kohler

An undergraduate thesis submitted to the Department of Computer Science of Harvard University in partial fulfillment of the requirements for the degree of Bachelor of Arts in Computer Science

Harvard University
Cambridge, Massachusetts
April 3, 2020

Abstract

Serverless computing provides a simple model for executing computations in response to triggers. In many serverless deployments, these computations are executed within virtual machines for stronger isolation and fairness guarantees. As a result, the time required to boot a new virtual machine is a significant contributor to the latency of a serverless application. To minimize the latency introduced by the boot process, we propose an alternative paradigm in which new virtual machines are launched by *flash-cloning* a pre-booted machine image. To assess this scheme, we have created HyperFork, a flash-clone implementation built on top of KVM and the `kvmtool` virtual machine monitor. HyperFork duplicates the full machine state, including memory, disk contents, and CPU state, to the new virtual machine. For the contents of memory and disk, our implementation leverages copy-on-write optimizations exposed by the Linux kernel. We demonstrate that HyperFork reduces virtual machine cold-start times by up to 98.4% while increasing throughput by 46%–502% in serverless workloads. We discuss the optimizations employed to achieve this performance along with security and correctness implications of our approach.

Contents

1 Introduction

Serverless computing provides a simple model for performing computations in response to triggers. In response to an incoming event, such as a web request, a serverless framework spawns an independent instance of a preregistered handler to process the event. For example, the framework may launch a transcoding job in response to a video upload, or compute a response after receiving an HTTP request. This new instance need not run on the same machine as prior instances. As additional events arrive, the framework can scale arbitrarily by placing jobs on more machines. Thus, serverless deployments can make use of heterogeneous infrastructure to scale to the demands of an application in real time. Cloud providers, such as Amazon Web Services, offer serverless computing as a means of scaling workloads arbitrarily while charging customers only for the resources consumed. In this way, serverless computing allows a workload to reach arbitrary scale without requiring the customer to reserve capacity.

In serverless settings, workloads are often encapsulated in virtual machines (VMs) to achieve isolation and fairness guarantees. When an event arrives, a new VM is booted, the content of the event is provided to the VM, and the VM executes the workload, returning results when complete. While this provides a convenient encapsulation for jobs, the necessity of booting a new VM on each request adds significant latency to serverless requests. Simply booting a stripped down version of the Linux kernel can take hundreds of milliseconds [1]. This makes the VM encapsulation less suitable for latency sensitive workloads such as serving a website or providing real-time analytics, and makes it difficult to use frameworks or applications requiring substantial initialization in serverless jobs.

We propose HyperFork, a virtual machine manager (VMM) based on `kvmtool` and KVM, as a solution to this problem. Instead of booting a new VM in response to each event, we boot a single virtual machine, suspend its state after booting and initializing, and then *flash-clone* it in response to an incoming event. The flash-clone operation duplicates all the state of the VM and VMM process. This includes the contents of guest memory and disk, guest CPU state, and hardware events pending in the VMM. After a flash-clone operation, we are left with two identical, independent, booted copies of the virtual machine, one of which can be resumed to handle an event without any significant latency. The other remains in its suspended state to be flash-cloned when another event arrives. By booting a single virtual machine, suspending it, and flash-cloning it in response to events, we can begin processing new events with only the latency of the flash-clone operation. Fundamentally, the flash-clone operation is based on the Linux `fork` syscall, with significant userspace additions to efficiently preserve and restore KVM state.

The userspace additions are necessary as the KVM data structures in the kernel are not fork-aware, becoming inoperable in the child process post-fork. Therefore, HyperFork must extract all KVM virtual machine state prior to the fork, create fresh KVM structures in the child, and then install the saved state into the new virtual machine. Furthermore, the multithreaded architecture of VMMs interacts poorly with `fork`. Many VMMs, such as `kvmtool`, utilize one thread per guest

CPU, along with threads for workqueue processing, emulating devices, and handling IPC. After a `fork`, only the forking thread will survive. Because of this, HyperFork must properly suspend the other threads at a consistent point prior to forking and recreate them in the child process. We will discuss the challenges of duplicating KVM and VMM state performantly and evaluate a series of solutions.

HyperFork achieves significant performance gains over the naive cold-start method employed by other VMMs. We find that launch times can be reduced by a factor of up to 60, while throughput on a realistic serverless workload can be increased by a factor of more than 6. A flash-clone operation further enables frameworks requiring significant initialization to run efficiently in a serverless environment. As a proof of concept for this point, we show that HyperFork enables serverless jobs to use local Spark applications that were previously latency-prohibitive. Our implementation of HyperFork is available at colavitam/hyperfork-kvmtool.

We first explore related work on virtualization, serverless computing, and duplication of VM state, then provide an overview of the existing implementations used to build HyperFork. Next, we discuss the implementation of HyperFork and other components of our benchmark suite. We then describe in detail the internals of KVM's memory management system and its performance implications for HyperFork. We then propose a series of benchmarks and summarize quantitative results for flash-clone performance. Finally, we discuss the current state of HyperFork, explain its utility as part of a serverless framework, and assess security and correctness concerns with our approach.

2 Related Work

We begin by summarizing the past research upon which HyperFork builds, considering work on virtualization methods, cold-start latency, duplication of virtual machines, and live migration. We then discuss specific implementations useful to HyperFork.

2.1 Existing research

2.1.1 Virtualization

Virtualization is a method of emulating the hardware resources of a bare-metal environment within software. These emulated environments, known as virtual machines, act as independent entities on the machine. Through a variety of techniques, a bare-metal machine simulates the behavior of hardware, including the CPU, MMU, and PCI devices, so that workloads which normally require an entire machine can be run in an isolated software environment instead. A hypervisor is a piece of software that runs with full privilege on the host machine to manage and isolate a set of guests. Virtualization can emulate either the architecture of the bare-metal system, or an unrelated architecture.

One common method of virtualization is known as *full virtualization*, which allows allows guests to run unmodified as the full range of processor behavior is supported in the virtual machine environment. For architectures that do not match the host machine, this requires emulators that implement the full behavior of the processor. QEMU is one such emulator, providing support for a wide array of architectures and hardware [2]. For architectures that do match the host machine, privileged instructions must be properly trapped or rewritten to ensure that the virtual machine cannot access protected host state.

As an alternative to full virtualization, many hypervisors support *paravirtualization* [3], a technique employed in Xen [4]. Paravirtualization enables the hypervisor to efficiently emulate the native architecture of the host system. In a paravirtualized environment, the guest operating system must be modified to replace privileged CPU instructions and hardware interactions with calls to the hypervisor (known as *hypercalls*). In most settings, paravirtualized environments do not support privileged instructions that would need to be trapped by the hypervisor. Because of this, the guest operating system can run directly on the processor without needing to trap into the hypervisor. Paravirtualization can thus provide more performant virtualization by better mapping to the abstractions of a virtual machine instead of raw hardware.

Paravirtualization is also used to improve the performance of interacting with hardware. Under full virtualization, the devices exposed to the guest are often emulated versions of real hardware devices. Communication interfaces that are efficient to implement in hardware may not be as efficient to emulate by trapping. Therefore, we would like to expose devices that are both easy to use in the guest and efficient to virtualize in the host. One such class of devices is *paravirtualized devices*, in which the guest is aware of a protocol to communicate with the device through the hypervisor. This requires the guest to be virtualization-aware, but can simplify hypervisor implementation and improve device performance.

`virtio` is a general protocol used for paravirtualized devices, intended for use by Linux guests [5]. It uses a ring buffer allocated in the guest for communication with the host and specifies architecture-specific mechanisms for efficiently notifying the hypervisor of pending messages. Similarly, the hypervisor can efficiently communicate to the guest by modifying this buffer and issuing an IRQ. `virtio` implementations exist for block devices, network devices, random number generators, consoles, PCI devices, and more.

For virtual machines using the native architecture, we aim to run their instructions directly on the CPU for better performance. Note that because privileged instructions can be used by the guest, the hypervisor must trap these instructions and emulate them to prevent the guest from accessing sensitive hypervisor state. Older hypervisors, such as early versions of VMWare [6], achieve this by *binary translation*, in which privileged instructions in the guest are replaced with code that properly updates the virtual machine state while respecting the guest-host isolation boundary. Unfortunately, binary translation is difficult to implement and less performant than paravirtualization in practice [6]. VMWare has since deprecated this approach [7].

Nearly all modern hypervisors trap privileged instructions using dedicated CPU support. Many modern processors, including those from Intel and AMD, provide hardware-level support for virtualization, including dedicated instructions and data structures for accessing guest state and switching between host and guest contexts [8]. These allow the host machine to easily trap privileged CPU instructions and intercede on address translation and device operations where appropriate. The Linux kernel exposes this kind of virtualization through the KVM kernel module [9], which is used from userspace by virtual machine monitors such as QEMU (in native KVM mode) [2], `kvmtool` [10], and Firecracker [11]. KVM requires CPU virtualization support, as it does not implement full paravirtualization or binary translation [9].

2.1.2 Optimizing cold-start latency

Reducing cold-start latency, the amount of time required to boot a virtual machine for our serverless job, is of utmost importance to serverless deployments. Cold-start latency increases the latency of serverless jobs and can be a significant drain on machine resources. In the case of serverless deployments based on virtual machines, cold-start latency is usually dominated by the boot time of the virtual machine image [11].

One straightforward method of reducing cold-start latency is to directly reduce the amount of time needed to boot the VM operating system. As VMMs often simulate a vastly smaller set of hardware profiles than those available on a bare-metal machine, kernels for the virtual machine image need not support the wide variety of devices they are usually compiled with. By removing these unneeded device drivers and features from the kernel, we can significantly decrease the boot time. This was the approach taken by Amazon in their Firecracker VMM, which supports only one hardware configuration. Deployment images for Firecracker include only the drivers and kernel features necessary to support this configuration, leading to reported boot times under 150 ms [11].

Unikernels take an alternative approach to removing unnecessary operating system features. In a unikernel, operating system features are treated as dependencies that can be included in the job if required. When the job is compiled, only the set of kernel features necessary to satisfy this set of dependencies is included, leading to an extremely minimal kernel [12]. However, explicitly representing kernel dependencies in an application can add significant overhead to a project or constrain the languages and libraries that can be used for the implementation. Denali OS takes a similar unikernel-based approach [13].

Some serverless deployments evade boot latency by making use of warm-starts, in which virtual machines are either pre-booted or reused after completing a job. This allows us to incur the overhead of booting a virtual machine prior to the request. However, we must still incur cold-start latency when the serverless provider does not have a VM for our function loaded in memory. Functions are usually evicted from memory after our job has not been invoked for several minutes [14]. Some providers, such as AWS, allow customers to pay to keep a certain number of function instances warm-started [15].

Other methods avoid boot time entirely by moving the abstraction boundary above the machine level. Containerization technologies such as Docker [16] use abstractions provided by the Linux kernel to isolate themselves from other processes on the system. These abstractions, known as `cgroups`, attempt to emulate the isolation and fairness guarantees of a virtual machine without the overhead of running an independent kernel on the system. In many cases, however, containers provide weaker fairness guarantees than virtual machines and run a greater risk of guest-to-host breakouts due to their higher abstraction boundary [17]. Common container implementations, such as Docker, have experienced such vulnerabilities in the past [18].

The use of libraries and frameworks requiring extensive initialization can also contribute to the cold-start time of a virtual machine. For example, loading the JVM and Spark or preparing local databases can add seconds to the initialization time of a virtual machine. In the absence of warm-start, this initialization must be done for every request that arrives, making the use of such frameworks expensive and slow. If we perform this initialization ahead of time with warm-start, we must maintain enough virtual machines to handle the peak concurrency for our function. As a primary goal of serverless programming is to scale dynamically for any level of concurrency, maintaining the peak number of instances for every function registered on the platform is infeasible. Flash-clone primitives solve this problem by enabling virtual machine state to be saved after initialization is complete. Solutions using containers are significantly more complex and require application-specific implementations or shared resources, complicating the isolation boundary [17].

2.1.3 Flash-cloning

Flash-cloning, our method for reducing cold-start latency, was previously explored in the Potemkin Virtual Honeyfarm [19], a system that enabled a large number of honeypots to be hosted on a single machine. Flash-cloning duplicates existing virtual machine state, creating an independent VM without repeating the boot process. Potemkin implements flash-cloning within the Xen hypervisor and aims to maximize VM density and resource sharing instead of startup latency. Our implementation instead implements flash-cloning atop KVM using the `fork` system call. This allows us to exploit the copy-on-write functionality provided by the Linux kernel.

2.1.4 Live migration

Flash-cloning also bears some similarity to virtual machine live migration [20, 21]. Live migration allows a running virtual machine to be transferred from one physical host machine to another with minimal downtime. As this requires duplicating the virtual machine state, many of the techniques used for saving virtual machine state that we exploit in HyperFork were pioneered by live migration research. Live migration usually emphasizes minimizing downtime and optimizing network bandwidth, whereas our approach prioritizes minimizing latency.

SnowFlock, another project that proposes forking a virtual machine, combines the notion of flash-cloning and live migration [21]. SnowFlock is built atop Xen and focuses primarily on efficiently transmitting virtual machine state over the network. Its design caters to distributed computation that uses forking as a primitive for distributing a computation to another machine. HyperFork, in comparison, focuses on optimizing flash-clones within a single machine by building atop the Linux `fork` operation.

2.2 Relevant implementations

2.2.1 KVM

KVM is a Linux kernel module which provides support for running guest virtual machines within Linux processes. As native virtualization support on most architectures requires privileged instructions [22], KVM safely exposes this functionality to userland. To ensure that all guests stay properly isolated from the host and each other, KVM isolates virtual machines within Linux processes and defines an interface by which host processes can access the state of their VMs. The VMM communicates with KVM via a set of `ioctl`s performed on file descriptors created using the KVM API. A KVM virtual machine is structured as a set of vCPUs, each corresponding to a single core of the VM's virtualized CPU.

Each VM is hosted within a standard Linux process, containing both VMM management components and the state of the running guest. Threads of the VMM process operate in either user mode (like a standard Linux process), or guest mode. In guest mode, the thread is emulating one of the VM's vCPUs through KVM. To enter guest mode, the VMM performs an `ioctl` on the file descriptor of the vCPU it wishes to emulate. When the vCPU performs I/O or other events that must be processed by the VMM, the thread returns to user mode, continuing execution directly after the `ioctl`. This allows us to structure our VMM as a set of threads corresponding to each vCPU in the VM [9].

VMMs often contain CLI or HTTP-based management programs for administrators to control VMs. For our implementation, the userspace VMM components are implemented by `kvmtool`, but the structure of KVM-based VMMs is conserved across a wide variety of implementations including Firecracker [11].

2.2.2 kvmtool

`kvmtool` [10] is a VMM implementation for KVM with the minimal functionality required to boot a fully functional Linux kernel with basic virtualized devices. Supported devices include block, network, filesystem, balloon, hardware random number generation, and console virtual devices, along with a legacy 8250 serial device. `kvmtool` is provided as an alternative to heavier VMM solutions such as QEMU [2], which supports a wide range of legacy devices and guest configurations. `kvmtool` is a good basis on which to build HyperFork, as it is both minimal and efficient, providing an excellent environment for serverless workloads—which rarely depend on device support.

`kvmtool` also offers a very similar set of functionality to Firecracker, the VMM used to power Amazon Lambda. By using simple hardware implementations, startup times and memory footprints are minimized. For increased security on shared tenancy machines, Firecracker uses strict containerization schemes on top of virtualization-based sandboxing and is implemented in Rust. Firecracker offers only a RESTful API for VM creation and management. In contrast, `kvmtool` uses a simple command line interface and IPCs to manage guest VMs.

To virtualize efficiently, `kvmtool` makes use of a number of threads for managing each vCPU and emulated device. Userspace bookkeeping data structures hold file descriptors which are used to access internal VM state maintained by the KVM kernel module. When the virtual machine is started, `kvmtool` creates a thread for each class of device, including the terminal, 8250 serial console, block devices, and other `virtio` devices. `kvmtool` also creates one thread per vCPU that proceeds in a loop, entering guest mode, then handling any IO requests or interrupts that may arise. To implement HyperFork atop `kvmtool`, all of this state must be properly duplicated in the child process.

3 Design

We now describe the design and implementation of the HyperFork hypervisor. We begin by detailing how HyperFork implements the flash-clone operation and exposes this functionality to guests. Next, we describe the process of optimizing HyperFork for the KVM MMU. In doing so, we explain the structure of the KVM MMU and how various memory operations are implemented. With this understanding, we provide a performance optimization for general guests and a series of optimizations specific to HyperFork.

3.1 The HyperFork hypervisor

HyperFork is based on the virtual machine monitor `kvmtool` (§2.2.2). It supports only basic hardware used in the Linux booting process and hardware that can be efficiently virtualized through `virtio` protocols. We selected `kvmtool` as the basis for HyperFork as it cleanly encapsulates most details of virtual machine state and features a minimal and readable codebase.

3.1.1 KVM state duplication

As a fundamental goal of HyperFork, we wish to efficiently duplicate the entire state of a virtual machine after boot and after important software (e.g. Java) has initialized. Note that for VMMs based on KVM, this state is split between usermode components—the threads backing

CPUs, workqueues corresponding to devices, bookkeeping file descriptors, and backing memory—and kernel components—including vCPU state (the state of each processor in the guest) and host pagetables (which define the mappings between guest physical and host physical addresses). In order to fully duplicate the VM state, both the kernel and usermode state must be duplicated.

KVM uses an API based on file descriptors. When a process wishes to create a new KVM virtual machine, it first opens the KVM device, obtaining a file descriptor on which it can perform `ioctl`s. One such `ioctl` creates a new virtual machine instance (represented by a fresh file descriptor) for the process. From this virtual machine file descriptor, the VMM can then create file descriptors corresponding to each vCPU and KVM device. Each of these file descriptors is backed in the kernel by implementations of the KVM-specific `ioctl`s along with state specific to the virtualization primitive. For example, a VM file descriptor contains information about the memory mappings of the virtual machine and various state shared between processors. A vCPU file descriptor, on the other hand, contains the control block used for the bare metal processor's virtualization extensions. All kernel KVM state is reachable from the state attached to these file descriptors.

On Linux, the standard method for creating a new process with usermode state identical to another is to fork the original process using the `fork` system call. This duplicates the address space of the original process while sharing the open file descriptors between the two processes. While this is the basis of our VM duplication mechanism in HyperFork, it unfortunately falls far short of duplicating all necessary state. `fork`, while an excellent primitive for duplicating the contents of an address space and sharing simple file descriptor context, has many shortcomings that render it a poor choice for more complex file descriptor APIs and multithreaded environments. In fact, we encounter many of the shortcomings enumerated by Baumann, et al. in their critique of the fork paradigm [23].

Specifically, `fork` preserves only the calling thread in the child process. That is, if the parent process contains several threads that are running at the time of the fork, only the thread which invokes `fork` will survive in the child. This poses two major problems. First, we must restart these threads and ensure they resume work at a consistent point. Second, we must ensure that the other threads are not performing any kind of atomic operation during the fork. Suppose, for example, another thread is holding a lock at the time of the fork. When the child process resumes, the thread holding the lock will no longer exist, but the lock will remain in a locked state. It is undefined behavior to unlock this lock from a thread other than the one that locked it, which no longer exists [24]. Similarly, if another thread has dequeued a task from a global workqueue and is storing it on its stack, the task will be lost when the thread is recreated in the child. Thus before forking we must ensure that each non-forking thread is at an interruptible state and not holding locks.

Furthermore, `fork` does not enable us to transparently copy kernel state. While forking shares file descriptors—which themselves contain the KVM state—between the parent and child process, the parent and child file descriptors refer to the same instance of KVM state in the kernel. That is, the file descriptors in the child and parent both point to state corresponding to a single, shared (not duplicated) virtual machine. To avoid synchronization problems arising from this case, KVM prohibits the use of `ioctl`s on file descriptors originating from other processes, including a parent. Therefore all KVM file descriptors become unusable, and their associated state is inaccessible in the child.

To overcome this limitation, we must recreate the file descriptor structure in the child process and initialize it with the state of the parent virtual machine. We explored three potential solutions to this problem:

KVM kernel module extensions. Originally, we attempted to modify the KVM kernel module to automatically duplicate this state when a process is forked. This required extensive modifications to the KVM implementation, many of which were platform specific and extremely difficult to verify. After several weeks of effort, we were able to duplicate the MMU management state corresponding to the virtual machine, but then abandoned the idea after encountering the minimally documented VMX state necessary to clone vCPUs. Continuing down this path would also have required us to update memory mappings corresponding to the vCPUs, which are used by the VMM to extract information from the processor when KVM exits guest mode. Overall, we assessed the effort and risk of error of this approach to be extremely high and insufficient to justify any gains in performance. Profiling revealed that roughly 53% of the time needed to fork was consumed duplicating KVM state. As many of the routines to duplicate KVM state from userland are identical to those that would be invoked from the kernel, profiling results suggest at most 8% of fork time could be saved by an in-kernel solution.

Parent-to-child IPC. Another approach we considered was to use IPC to transmit the context corresponding to each file descriptor from the parent to the child after the fork operation. The child could then regenerate each of the file descriptors by opening the KVM device again and creating fresh VM, vCPU, and device file descriptors. Once the structure has been recreated, the child can initialize each of the file descriptors states' with the state received from the parent. This approach appeared to be feasible, but we rejected it in favor of the final approach which avoids the complexity and overhead of IPC.

Pre-fork state extraction. The approach we ultimately adopted is to extract all relevant kernel state from the file descriptors before the fork is initiated. Extraction of state is performed using the standard KVM API, saving vCPU, interrupt, clock, and timer state in the memory of the parent process. Once the VM state is saved in its entirety, we initiate the fork process. As the memory of the parent and child will be identical after the fork, the child retains access to the recorded state. The child then regenerates the virtual machine file descriptor structure as in the IPC approach and immediately initializes their states using the data saved in memory. The state can then be freed and the virtual machine resumed.

3.1.2 Pre-fork and post-fork routines

Pre-fork. In summary, we perform the following sequence of steps before forking:

1. All vCPUs in the virtual machine and their corresponding threads are paused. This ensures that our vCPU state is consistent in the child.
2. vCPU registers, interrupt state, interrupt configuration, clocks, and counters are extracted using the KVM API and saved.
3. All mutexes that may be held by other threads (such as `virtio` and device workers) are acquired to ensure they are not in a locked state at the time of fork.

Note that it is not necessary to save the memory of the virtual machine, as this is present in our address space and will be duplicated automatically by `fork`. We next perform the fork operation.

Post-fork. In the child, we perform the following sequence of steps to restore VM and thread state:

1. Fresh file descriptors for the KVM device, virtual machine, vCPUs, and devices are acquired.
2. All recorded state is restored to the fresh file descriptors using the KVM API.
3. All `eventfd`s used for signaling between threads are recreated.
4. New threads corresponding to each vCPU, device, and worker are created.
5. All mutexes locked in the pre-fork routine are released.
6. All condition variables are replaced.[1]

At this point the child can resume virtual machine execution.

[1]Unfortunately, because many condition variables contain an internal mutex that cannot be separately locked, it is difficult to guarantee that they are left in a consistent state after the fork. We encountered deadlocks due to permanently locked condition variables when we did not regenerate them.

3.1.3 Guest-to-host communication

After booting and loading any environments necessary to perform its job, a virtual machine must signal to HyperFork that it is ready to be held in stasis or forked as appropriate. To do so, we send signals through x86 I/O ports, which are natively supported in userland by Linux and trap directly into the hypervisor. The guest may send a *fork* signal, indicating that it has completed initialization and may be suspended or forked. The *done* signal terminates the virtual machine from the hypervisor and frees its resources, without requiring teardown in the guest kernel. We extend this communication system with additional signals for use in benchmarks and profiling.

Each signal is implemented in a statically linked executable that is made available within the guest root partition images.

3.1.4 Security considerations

Cloning the full state of the guest virtual machine has potential security implications that must be addressed. In a serverless deployment using HyperFork to generate virtual machine instances, each new virtual machine will have kernel state identical to the parent. This includes the state of the kernel entropy pool along with any materialized randomness in the guest, such as the memory layout of the kernel and processes.

Entropy pool. A common entropy pool between new guests jeopardizes the security of random number generators used in the guest and thus any security primitive that relies on their randomness for security. Given the same PRNG state in the kernel, the same random numbers will be provided to kernel and userland routines that request them across different guests. While CPU jitter and noise would eventually cause the entropy pools in different guests to diverge [25], the period of time after flash-cloning during which they match closely is a potential security vulnerability. The random numbers generated across different guests may be identical or biased in a way that invalidates security assumptions of downstream cryptographic primitives.

To correct for this weakness, we must reseed the entropy pool in the newly cloned guest. Fortunately, Linux offers an easy facility to achieve this: writes to `/dev/urandom`. Immediately after a fork is performed in the guest, the guest will read random bytes from `/dev/random` and write those bytes back to `/dev/urandom` before resuming guest programs. While reads from `/dev/urandom` are backed by the kernel PRNG and entropy pool, reads from `/dev/random` must come from an entropy source on the system [25]. Typically, this originates from IRQ noise, CPU noise, jitter, or dedicated hardware RNGs on the system. Fortunately, because we are working in a virtualized environment, we can supply each guest with an independent source of randomness by exposing a `virtio` hardware RNG backed by the kernel PRNG in the host. Thus, when each guests reads from `/dev/random`, it receives independent randomness from the host which is then used to reseed its own entropy pool. This ensures that random number generation in the guests is independent after the flash-clone.

Materialized randomness. Materialized randomness, or state that depends on randomness obtained before the guest was cloned, is much more difficult to address. Certain materialized randomness, such as the layout of the kernel in KASLR, is fixed during the boot process and cannot be modified after the flash-clone [26]. This is also true of ASLR in processes that were created prior to the flash-clone. We discuss these concerns further in §5.1.

3.2 Optimizing memory management

To better understand the performance characteristics of HyperFork's flash-clone operation, we now discuss the implementation of the x86-64 MMU on KVM. While MMU support is provided transparently by KVM, our flash-clone operation must duplicate the parent's MMU state in the child. As described in §3.1.1, we accomplish this by saving parent state prior to a `fork` and restoring it in the child. This implementation is correct, but while benchmarking we found that a significant amount of time in the `fork` syscall was spent in KVM MMU routines. We will explain the structure of the KVM MMU, why it degrades the performance of `fork`, and how we have addressed these problems in HyperFork. Furthermore, we will address general concerns of efficiently virtualizing an MMU on x86 and the interaction between copy-on-write and the KVM MMU.

On modern x86-64 architectures, KVM makes use of Intel and AMD's two-dimensional pagetable implementations, which will be discussed in detail in §3.2.1. Our description is based on several pieces of Linux kernel documentation, along with careful inspection of the kernel codebase [27, 28].

In order to keep track of the various notions of virtual and physical addresses, we define the following terminology, consistent with naming schemes within KVM:

- **Host physical address (HPA)**: The physical address backing both the host virtual addresses and the guest physical addresses. This is also known as the machine address.
- **Host virtual address (HVA)**: A virtual address used in the address space of the host VMM process. These are used by the VMM process to access the memory of the guest directly.
- **Guest physical address (GPA)**: A physical address used in the guest. Each guest physical page is logically backed by a single host physical page, though the host kernel may share pages with copy-on-write semantics.
- **Guest virtual address (GVA)**: A virtual address used in the guest. These are used within the VM to access memory based on the currently loaded guest pagetable. The virtual machine is free to modify mappings between guest virtual addresses and guest physical addresses by installing or modifying pagetables.

- **Guest pagetable**: Specifies the mappings between guest virtual and guest physical addresses. All pagetables contain pagetable entries (PTEs).
- **Host pagetable**: Specifies the mappings between guest physical and host physical addresses.
- **Shadow pagetable**: Specifies the mappings between guest virtual and host physical addresses. It is generated from the guest and host pagetables and installed in place of the guest pagetable on architectures without two-dimensional paging.
- **VMM pagetable**: Specifies the mappings between host virtual and host physical addresses.

3.2.1 Two-dimensional paging

On bare metal x86 systems, virtual memory is implemented through a pagetable structure installed in the `cr3` register. This pagetable provides a partial mapping from virtual addresses to physical addresses and is traversed by the processor when virtual address translation is required. Supervisor-level code can change these mappings by installing a new pagetable or modifying the current one.

A primary goal of x86 virtualization is to virtualize this same memory management functionality for unmodified guests running in the virtual machine. Without dedicated support, this appears challenging. The guest will construct a pagetable mapping guest virtual to guest physical addresses, then attempt to install this pagetable in `cr3`. Allowing the guest to install its pagetable into the processor's `cr3` register would be problematic, as guest physical addresses do not necessarily match host physical addresses. Furthermore, allowing the guest to have full control over its paging structures could allow it to access memory belonging to the host, violating isolation. Therefore, we must either trap these modifications to `cr3` and guest pagetables, or rely on additional support from the processor.

Memory virtualization features are known as two-dimensional paging (TDP). TDP simplifies this problem by offering both host and guest pagetables at the architecture level. The guest pagetable provides translations between guest virtual addresses and guest physical addresses. The host pagetable provides translations between guest physical addresses and host physical addresses. Under this scheme, the guest can fully control its own virtual address mappings by constructing pagetables and installing them into the guest `cr3`. When the guest physical addresses are resolved, the processor will attempt to map these guest physical addresses to host physical addresses using the host pagetable. If this translation fails, page-fault handlers will be invoked, allowing the host to handle the fault as appropriate.

On architectures without two-dimensional paging, emulating virtual addressing in guests requires special support from the hypervisor. KVM addresses this problem with shadow pagetables. Shadow pagetables contain mappings between guest virtual addresses and host physical addresses, and are generated by KVM whenever a new pagetable is installed or an existing one is modified. This one-dimensional shadow pagetable is installed in `cr3` in place of the guest pagetable

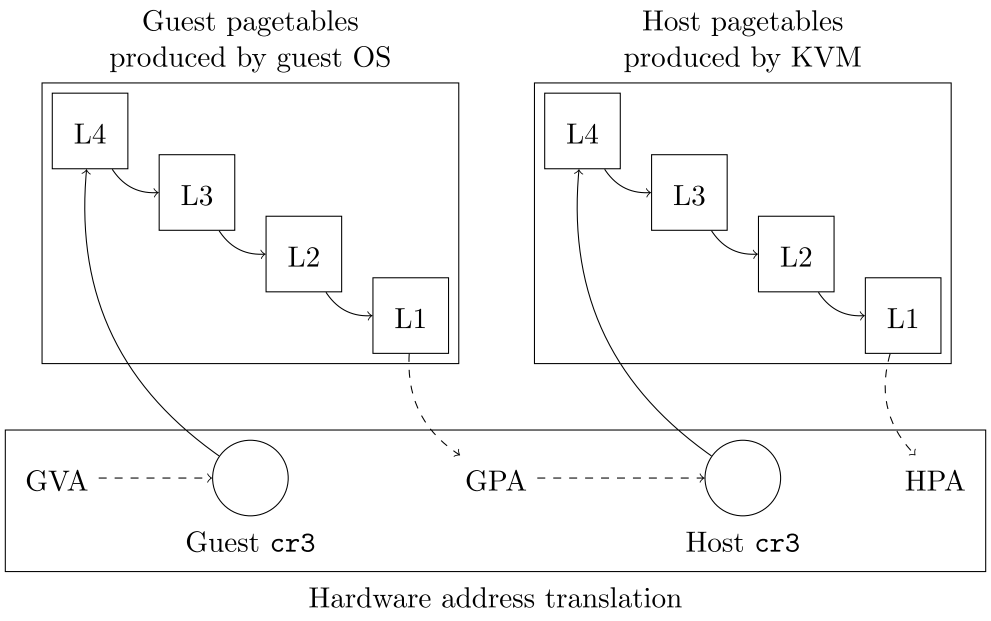

Figure 1: Address translation in TDP mode

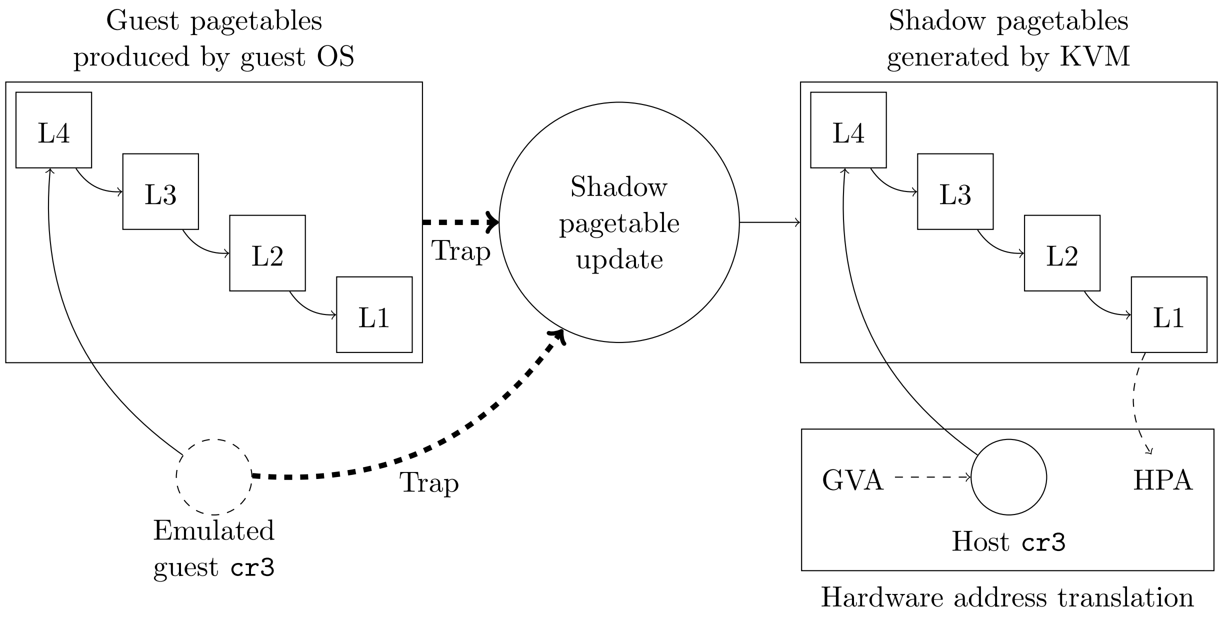

Figure 2: Address translation without TDP

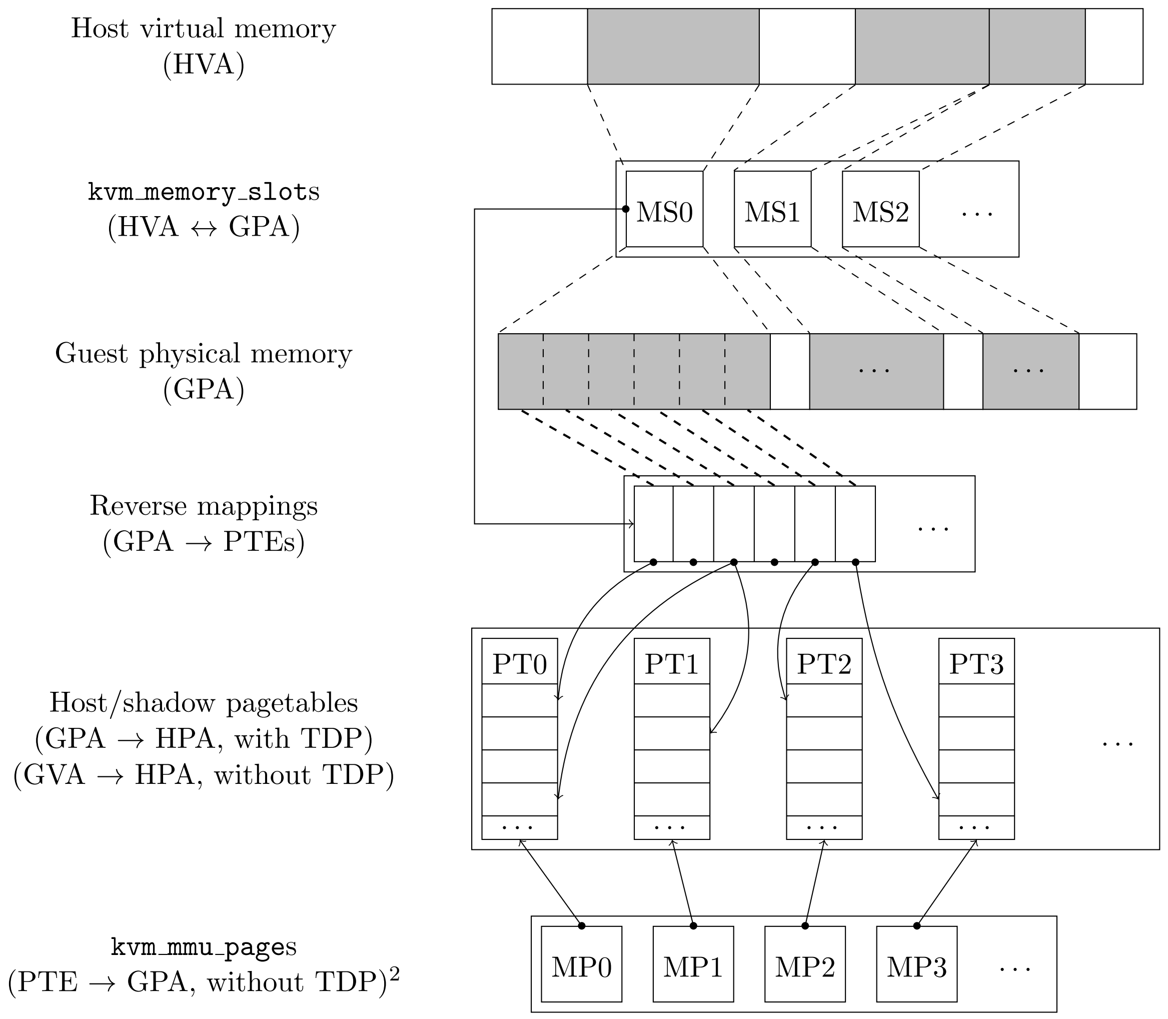
Host virtual memory (HVA)
kvm_memory_slots (HVA ↔ GPA)
MS0
MS1
MS2
...
Guest physical memory (GPA)
...
...
Reverse mappings (GPA → PTEs)
...
Host/shadow pagetables (GPA → HPA, with TDP) (GVA → HPA, without TDP)
PT0
PT1
PT2
PT3
...
kvm_mmu_pages (PTE → GPA, without TDP)[2]
MP0
MP1
MP2
MP3
...

Figure 3: Relationship between KVM memory management structures, with reverse mapping for one-dimensional paging

to provide translations. While this allows us to use the processor's standard translation support for virtual addresses, it imposes significant overhead, as pagetable installations and modifications must be trapped. Generating shadow pagetables requires traversing and translating the entire guest pagetable. On our benchmarks, disabling TDP increased runtimes by factors of up to 3.6.

3.2.2 Important KVM MMU structures

We now discuss the primary structures for memory management in KVM. These structures will play a critical role in the memory operations and translations described below. The relationship

[2]When TDP is disabled, the `kvm_mmu_page` points to a shadow pagetable mapping guest virtual addresses to host physical addresses. In this case, it also maintains a parallel pagetable mapping the same guest virtual addresses to guest physical addresses. This is not illustrated in the diagram.

between these structures is summarized in Figure 3.

The kvm_memory_slot structure. The `kvm_memory_slot` structure describes the regions of memory allocated to the guest by the host VMM. When the VMM process initializes the VM, it provides regions of its virtual address space (acquired through `mmap` or similar) to KVM to back the guest's physical memory. Each of these regions is tracked with a single `kvm_memory_slot` structure, which contains its host virtual address, guest physical address, and size. Therefore the `kvm_memory_slot`s specify the correspondence between host virtual and guest physical addresses. It also contains the reverse mapping, which we will discuss shortly.

When KVM wishes to iterate through the entirety of guest physical memory, it does so by iterating over the `kvm_memory_slot` structures to determine the range of mapped guest physical addresses.

The kvm_mmu_page structure. The `kvm_mmu_page` structure is the primary data structure for memory management in KVM. Fundamentally, each `kvm_mmu_page` points to a single pagetable page of the pagetable that will be installed in the host `cr3`. This is a host pagetable for two-dimensional paging, or a shadow pagetable for one-dimensional paging. Each host/shadow pagetable page has a single corresponding `kvm_mmu_page`. The `kvm_mmu_page` structure contains the the level number, permissions, and synchronization status of its referenced pagetable. Furthermore, for one-dimensional paging, it also caches the guest physical addresses corresponding to each pagetable entry. This is useful as the referenced shadow pagetable would only contain mappings from guest virtual addresses to host physical addresses.

Vitally, the role of the `kvm_mmu_page` structure changes when two-dimensional paging is enabled. With one-dimensional paging, each `kvm_mmu_page` corresponds to a single guest pagetable page, which may not be a linear range of guest physical addresses. This is known as a non-direct `kvm_mmu_page`. When two-dimensional paging is enabled, each `kvm_mmu_page` references a pagetable page for a linear range of guest physical addresses, corresponding to a linear range of host virtual addresses. This is known as a direct `kvm_mmu_page`. This distinction will be important when we discuss performance implications of the reverse mapping. When we say a `kvm_mmu_page` maps a set of addresses, we mean that it corresponds to a pagetable providing a mapping for that set of addresses. Therefore in one-dimensional paging, each `kvm_mmu_page` maps contiguous guest virtual addresses to host physical addresses. In two-dimensional paging, each `kvm_mmu_page` maps contiguous guest physical addresses to host physical addresses.

The correspondence between `kvm_mmu_page`s and `kvm_memory_slot`s is a bit complex. In direct mode, a `kvm_mmu_page` maps contiguous regions from the `kvm_memory_slot`s, but could contain more than one if a boundary point of the `kvm_memory_slot` falls within the interval of the `kvm_mmu_page`. In non-direct mode, there is no easy correspondence, as the `kvm_mmu_page` maps a contiguous region in the guest virtual address space, which could reference arbitrary portions of guest physical memory, and thus arbitrary `kvm_memory_slot`s.

The reverse mapping. In §3.2.3, we will discuss a series of operations that benefit from an efficient means of finding the set of pagetable entries in the host/shadow pagetables corresponding to a given guest physical address. With two-dimensional paging, our host pagetable maps from a linear range of guest physical addresses to host physical addresses, and thus the single mapping corresponding to the guest physical address can be found by a standard walk of the pagetable. With one-dimensional paging, the pagetable maps from a linear range of guest *virtual* addresses, any number of which may correspond to the desired guest physical address. Naively, we would need to traverse the entire pagetable structure to find all such entries. Note that the features of `kvm_memory_slot` we have described are not helpful here, as they only specify the correspondence between host virtual and guest physical addresses. We instead wish to locate host/shadow pagetable entries corresponding to guest physical addresses.

To address this problem, each `kvm_memory_slot` contains a *reverse mapping.* The reverse mapping contains an entry for each guest physical address mapped by the memory slot. Each entry contains a list of pointers to every host/shadow pagetable entry corresponding to that guest physical address. For two-dimensional paging, this is a linear mapping in which each list has a single element. For one-dimensional paging, each list can have an arbitrary number of elements. A reverse mapping exists for each of the four levels of the pagetable structure. That is, given any guest physical address, the reverse mapping provides pointers to the corresponding pagetable entries in each of the L1-L4 pagetables.

3.2.3 VM memory operations

We discuss how common memory operations are performed in the context of two-dimensional and one-dimensional paging.

Memory access from VMM process. Accessing memory from the VMM process requires a translation from host virtual to host physical addresses. This is handled by the VMM pagetable, which is installed when the VMM process is running. This behavior is identical to standard virtual to physical address translation on Linux.

Memory access from guest. The most common translation to be performed within the guest is a guest virtual address to host physical address translation when memory is accessed. With two-dimensional paging, this translation is invisible to KVM, as it is handled entirely by the processor. The performance implications of this approach will be discussed in §3.2.4.

In the case one-dimensional paging, KVM must generate a shadow pagetable when `cr3` is installed or the pagetable is updated. This shadow pagetable is installed on the CPU to provide translations for the guest. The translations are then performed transparently to KVM.

Page allocation from VMM process. When guest memory is accessed from the VMM process, the VMM reads the memory in the regions it provided to KVM. When the VMM process modifies this memory, modifications are visible to the guest. Likewise, when the guest modifies this memory, these modifications are visible to the VMM. This requires no special support; the VMM's virtual address mappings and the guest's physical address mappings are backed by the same physical pages, so updates will be visible to both transparently.

This does, however, require that the VMM pagetable and host/shadow pagetable be properly synchronized. For example, when the guest virtual address range is initially allocated, it may not be backed by physical pages if the region is zero-initialized. In this case, the kernel may use zero page sharing and copy-on-write semantics to allocate physical pages for the region when needed.

Suppose an unbacked guest physical page is modified in the VMM process. This will trigger a page fault to the host kernel to allocate the page and update the VMM pagetable accordingly. However, as the guest must also have access to this page, the kernel must also modify the host/shadow pagetable to map the corresponding guest physical address to the new host physical address. This requires translating from host virtual to guest physical addresses, which is performed by computing the offset of the host virtual address from the start of the corresponding `kvm_memory_slot` and using that as an offset from the guest physical address at which it is mapped.

Once the guest physical address is acquired, we have to update mappings in the host/shadow pagetable. In the case of two-dimensional paging, in which the pagetable maps from guest physical addresses, only one entry needs to be updated. In the case of one-dimensional paging, in which the pagetable maps from guest virtual addresses, several entries could correspond to a single guest physical address. If we wish to avoid traversing the whole pagetable structure, this requires a method of finding all pagetable entries corresponding to a given guest physical address. The reverse mapping of the corresponding `kvm_memory_slot` provides this.

Page allocation from guest. Suppose instead that an unbacked guest physical page is modified in the guest. In this case, a page fault will be triggered while traversing the host/shadow pagetable, which will lack a mapping for that guest physical address. The page fault will be handled by the KVM page fault handler, which will allocate the page and update the host/shadow pagetable accordingly, as described in the previous section. Again, as this mapping must be reflected to the VMM process, the fault handler must also install the mapping in the VMM pagetable. This requires a translation from a guest physical address (which triggered the page fault) to a host virtual address (where the corresponding VMM memory mapping exists). This translation is performed by traversing the `kvm_memory_slots` to locate the one which contains the guest physical address. The host virtual address of the base of the region is contained in this structure, and the final host virtual address can be computed based on the offset into the memory region.

Furthermore, note that with one-dimensional paging, we may have to update other entries in the shadow paging structure. This is because the guest physical page we are backing may have other corresponding guest virtual pages mapped in the shadow pagetable. This is done analogously to page allocation from the VMM.

Host write protection of pages. As a virtual machine's backing host physical pages are treated like normal kernel page allocations, they may be swapped out or write protected for copy-on-write. In either of these cases, KVM must update the mappings in the host/shadow pagetable to reflect the absence or modified permissions for the corresponding page. We again use the reverse mapping to locate the applicable pagetable entries.

Note that when the VMM process forks, all host physical pages mapped in the process will be write protected due to the kernel's use of copy-on-write optimizations. This includes the entire allocated guest physical address space and leads to significant performance implications.

3.2.4 General performance implications

Guest TLB misses. Note that the cost of a TLB miss in the guest is greater than the cost of a TLB miss in the host. In the host, a TLB miss triggers a pagetable traversal that walks the standard four level pagetable to resolve the physical address. In a guest with TDP, this pagetable walk must be completed first for the guest pagetable, producing a guest physical address, and then for the host pagetable, producing a host physical address. This requires walks of two four level pagetables, increasing the amount of time required for a guest TLB miss.

This is worsened by the fact that the guest pagetable is represented using guest physical addresses, which the processor must translate to host physical addresses while traversing the pagetable. Assuming an empty TLB, each pagetable access in the guest requires a full four level pagetable traversal in the host pagetable. Thus simply traversing the guest pagetable can require 16 host pagetable accesses. Our simple TLB miss has now resulted in a total of 4 guest pagetable accesses and 20 host pagetable accesses.

To help reduce this overhead, we can use a feature of many modern x86 processors, known in the Linux community as hugepages [29]. Hugepages allow page sizes other than the default 4KB in order to speed up pagetable walks and reduce TLB pressure. Specifically, when using hugepages to map a given virtual frame number to a given physical frame number, instead of using all four levels of the pagetable we can terminate the walk early. For example, a mapping terminating on the L3 pagetable would map a 2MB region, corresponding to the size of a mapping if that L3 entry had pointed to an L4 pagetable with a contiguous linear mapping. Similarly, a mapping terminating on the L2 pagetable would map a 1GB region, equal to the size of a region mapped by an L3 pagetable.

This decreases the number of pagetables which need to be walked to resolve a virtual address in this range, while also reducing the number of entries needed in the TLB to map a range of addresses. With 2MB hugepages, a single TLB entry can now map a region that would otherwise require 4096, significantly reducing pressure on the TLB and the number of misses we expect in

```
_do_fork ............................................ 100%
└── copy_process.part.36 ............................. 99%
    ├── __mmu_notifier_invalidate_range_start ........... 92%
    │   └── kvm_mmu_notifier_invalidate_range_start ...... 92%
    │       └── kvm_unmap_hva_range ........................ 92%
    │           ├── kvm_handle_hva_range ................... 54%
    │           │   └── kvm_unmap_rmapp .................... 54%
    │           │       ├── kvm_zap_rmapp .................. 37%
    │           │       │   ├── drop_spte .................. 18%
    │           │       │   └── rmap_get_first ............. 5.1%
    │           │       └── rmap_get_first ................. 2.6%
    │           ├── kvm_unmap_rmapp ........................ 26%
    │           ├── slot_rmap_walk_next ..................... 9%
    │           └── __x86_indirect_thunk_r14 ............... 3.4%
    └── copy_pte_range .................................. 6.8%
```

Figure 4: The performance trace of a fork operation (truncated at relevant depth). Note that `copy_pte_range` is the VMM pagetable duplication routine and `kvm_unmap_hva_range` is the host pagetable invalidation routine.

the guest.

3.3 Optimizing for HyperFork

HyperFork introduces a set of behaviors that the KVM MMU was not optimized for. Specifically, the flash-clone operation marks all pages backing guest memory as copy-on-write, requiring full invalidation of the host/shadow pagetable. Furthermore, these copy-on-write pages cause frequent page faults when the child VM resumes and modifies its memory. We now analyze the performance of these operations and propose HyperFork-specific optimizations.

Flash-cloning. In order to assess the cost of KVM's memory management on the HyperFork flash-clone operation, we profiled the kernel with Linux's `perf` tool during a series of HyperFork forks. This testing was performed with two-dimensional paging enabled to reduce memory management overheads. Within the kernel, the `fork` syscall consistently spent about 92% of its CPU time within KVM invalidation (write protection) routines. Specifically, when the kernel fork implementation dispatches to `copy_page_range`, it attempts to share the physical pages between the two child processes with copy-on-write semantics. This requires modifying both the VMM pagetable and the host/shadow pagetable in order to mark all pages as non-writable.

Note that only 7% of fork time is spent performing write protection in the VMM pagetable, compared to 92% spent on the host/shadow pagetable. We attribute this difference to the method of updating the two pagetables. The operations on the VMM pagetable are performed by directly traversing the entirety of the pagetable. For the host/shadow pagetable, KVM dispatches through `kvm_unmap_hva_range`, which walks the memory slots in sequence, traverses their reverse mappings, intersects them with the desired address range, and then follows the reverse mappings to the pagetable entries in the host/shadow pagetable, which are marked as non-writable.

80% of fork CPU time is spent inside `kvm_unmap_rmapp`, which updates pagetable entries and removes them from the reverse mapping. About 10% of time is spent solely on updating the reverse mapping iterator. Note that this is more than the total amount of time spent traversing the entire VMM pagetable. The reason for this slowdown relative to VMM pagetable invalidation is likely to be poor cache behavior. In the case of the VMM pagetable traversal, each pagetable page is traversed linearly, with minimal access to other parts of kernel memory. Thus, we are likely to only have to load each pagetable cache line from memory once. With the KVM reverse mapping, on the other hand, between every pagetable entry access, we return to the `kvm_memory_slot` struct to traverse the reverse mapping list. See Figure 5 for an illustration of this access pattern with an L4 pagetable. Even though the reverse mapping contains empty lists for all non-present pagetable entries, we must still traverse each entry in the reverse mapping to verify this. Furthermore, each present reverse mapping entry requires at least 4 function calls to process, adding additional overhead. Therefore, we access roughly the same number of reverse mapping entries as we would access pagetable entries by traversing the pages linearly. This causes poor cache behavior and likely leads to the slowdown.

In this way, the reverse mapping provides an efficient means of reaching the pagetable entries for a small number of guest physical pages. However, when we wish to access *all* of the pagetable entries, using the reverse mapping adds only overhead, as we access a roughly equal number of entries in the reverse mapping as we would traversing the pagetable directly.

There are two potential solutions to this inefficiency. One method is to separate pagetable traversal from reverse mapping traversal during invalidation. In other words, we will first traverse the host pagetable structure to invalidate all pagetable entries. We will then empty the reverse mapping directly. This avoids the need to follow the pointers in the reverse mapping and allows us to empty both structures in their in-memory order.

Alternatively, we can accomplish this from userland by deregistering (unmapping) the memory mappings backing guest physical memory prior to forking. This can be done using the standard KVM API for managing guest physical memory and leaves the memory regions in the VMM address space untouched. Upon deleting the mappings, KVM efficiently destroys the associated `kvm_memory_slots` (and therefore the reverse mappings) and invalidates the pagetables. Note that this is done directly instead of by traversing the reverse mapping. The mappings are then re-registered in the surviving child and parent.

We selected the latter approach for our implementation. This approach avoided any need to modify the host kernel and resulted in significant performance improvements. The only newly imposed overhead is the requirement to re-register the mappings in the surviving parent if we intend to resume the parent. This was found to take 0.25–4ms in the parent depending on memory

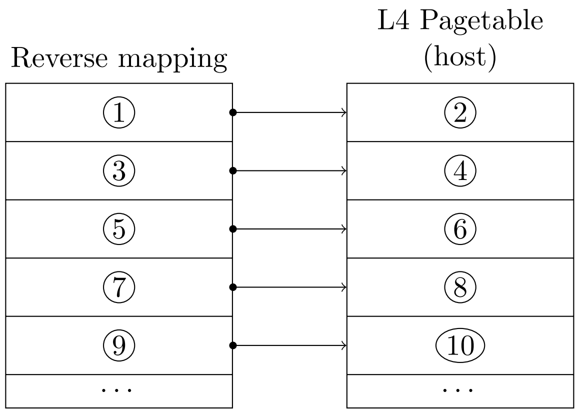

Figure 5: The alternating access pattern of the KVM invalidation routines during a fork for an L4 pagetable *with* TDP. Numbers denote the order of access. Note how the accesses alternate between the two disjoint structures. Each access requires at least two function calls.

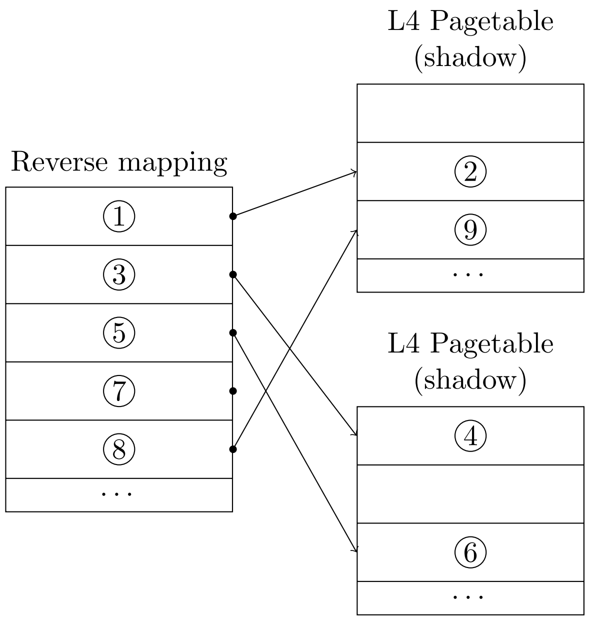

Figure 6: The scattered access pattern of the KVM invalidation routines during a fork for an L4 pagetable *without* TDP.

size. Furthermore, as the parent will be suspended indefinitely in the serverless application of HyperFork, re-registering the state in the parent is not necessary. We refer to this optimization as our *memory unmapping optimization.*

Host page faults. We further anticipate that host page faults will be an important component of guest performance both before and after a fork operation. When the memory backing the guest physical memory is initially acquired, much of it will be unbacked due to zero page sharing. Whenever these unbacked pages are modified by the guest, a page fault will be triggered while traversing the host pagetable. The host kernel must then back the page and update the pagetables as described above.

Furthermore, after a flash-clone, all pages backing guest memory will be marked copy-on-write, as they are shared between the child and parent. Whenever either guest modifies one of these pages, an analogous page fault will be triggered. Thus minimizing the overhead of these page faults plays an important role in ensuring performance before and after a flash-clone.

Again, hugepages may help to reduce these overheads. When the guest memory is backed by hugepages, page faults will be triggered for blocks of physical memory of size 2MB instead of 4KB. Thus, assuming all memory is written at some point, we expect page fault count to decrease by a factor of 4096 while the same amount of memory is copied. This will likely substantially reduce the overhead of the page faults.

Note, however, that this may negatively impact other workloads. Suppose that the workload writes to a sparse range of physical memory. In this case, smaller pages may result in less memory being duplicated by the kernel and greater performance. Thus there is a tradeoff between memory bandwidth required to duplicate the memory and the overhead of a page fault in the host kernel. The memory bandwidth overhead may result in especially large (1GB) hugepages introducing unacceptable stalls. We will investigate this tradeoff in §4.

4 Evaluation

To evaluate the performance of HyperFork, we wish to measure both the efficiency of the flash-clone operation itself and any performance degradation experienced by new virtual machines produced through flash-cloning. To do so, we first constructed a series of microbenchmarks to measure the time taken for a flash-clone operation under various conditions. Our clone benchmark was run under a series of host system configurations to assess how our implementation interacted with other common virtualization performance enhancements for KVM.

Furthermore, we constructed benchmarks to assess the overhead of various types of page faults experienced in the guest, both before and after a flash-clone. Finally, we performed benchmarks assessing the performance of CPU-bound and memory-bound computations in guests created by booting and flash-cloning. Using these benchmarks, we further measure throughput and latency.

4.1 Construction of virtual machines

Each benchmark was performed using a specially constructed root partition image for the virtual machine. The root partition images were produced using Docker by modifying standard images.

For fork and memory microbenchmarks, only a minimal root partition and utilities were required. For these, we created an image based off the official `busybox` image [30]. This image provides common Unix utilities through a single, statically-linked executable [31]. The contents of `/bin` are hard-linked to this single executable, which performs the correct function based on the invoked command name (`argv[0]`). This provides an environment with standard Unix utilities without any userland dependencies.

All benchmark images were augmented with the signal executables described in §3.1.3, which may be invoked from the benchmark scripts or executables to communicate with the hypervisor. When measuring fork performance, the fork signal suspends the parent and forks off a new child to resume work. In throughput benchmarks, it suspends state and repeatedly forks to model job dispatch. For timing operations in the guest, we introduce the *reset* signal, which prints the current values of the hypervisor's wallclock timers and resets them.

Within the images, we replace `/sbin/init`, the init process invoked after the kernel has booted, with a script to run our desired benchmark. Once the benchmark has completed, we send a done signal to terminate the virtual machine.

Once an image has been constructed using Docker, we create a corresponding Docker container, export the image to a `tar` archive, and then extract this archive onto a fresh `ext4` partition backed by a file. These partitions are then supplied as the root partitions for executing virtual machines. Note that this allows us to reuse the same kernel between separate benchmarks.

The scripts to generate the test images are available on GitHub at colavitam/hyperfork-bench.

4.1.1 Kernel configuration

All virtual machines used for testing were based on an unmodified Linux 5.0.7 mainline kernel. In order to minimize boot times, we employed configuration choices similar to Firecracker [11]. Specifically, we have removed all unused device drivers, disabled swap, removed unused IPC, minimized kernel tracing and CPU time accounting, disabled `cgroups`, and removed `initramfs/initrd` support. We believe this corresponds to a relatively minimal kernel for hosting a single job under a `kvmtool`-based hypervisor.

4.2 Benchmarks

All tests were performed on an `m5.metal` instance from Amazon Web Services, configured with 96 logical processors, 384 GB of memory, and version 4.14 of the Linux kernel. To assess the impact of hugepages and our memory unmapping optimization from §3.3, applicable benchmarks are run with all combinations of both options.

Fork benchmark. Our core microbenchmark assesses the performance of the flash-clone operation itself. This benchmark boots a Linux guest and immediately signals HyperFork to perform a flash-clone. The time taken for the new guest VM to resume is measured and reported as the flash-clone time. This is compared to the cold-start time of the virtual machine.

Memory benchmark. The memory microbenchmark assesses the overhead of page faults introduced by unbacked pages and copy-on-write semantics in a flash-cloned guest. This microbenchmark boots a Linux guest and immediately runs a C program. The program requests a region of memory slightly smaller than the available physical memory in the guest, and fills it entirely with pseudo-random contents (Loop 1). It then iterates over the region again, filling it with new contents (Loop 2). It then requests a flash-clone from HyperFork. When the guest is resumed, it fills the region again (Loop 3) and then one final time (Loop 4).

Each of these loops tests different conditions for guest pages. In Loop 1, the pages accessed in the guest are not backed by physical pages on the host due to zero page sharing. When the page is written to, a page fault in the host is triggered, allocating a backing page and updating the host pagetable accordingly. In Loop 2, the pages are backed in the host and no page faults are triggered. In Loop 3, two guests are sharing the same backing pages with copy-on-write semantics. Therefore, any write to the pages will trigger a page fault in the host, allocating a new page, duplicating the old page's contents, and updating both host pagetables accordingly. In Loop 4, the pages are backed in the host and no longer shared between the two virtual machines. The performance of each of these loops will provide insight into any performance degradation caused by flash-cloned guests using copy-on-write backing memory.

For this benchmark, 8GB was allocated to the guest, of which 7GB was traversed in each pass.

Java benchmark. The Java benchmark tests HyperFork's performance with more complex guest userland state initialized in the parent virtual machine. In this benchmark, a Linux guest is booted and then a simple Java application based on the Spark library is started. Once the JVM has started, the application preloads its required classes from the Spark library, then requests a flash-clone from HyperFork. After the flash-clone, a simple computation is performed using Spark. We use the multithreaded π estimation example included with the library.

This benchmark is meant to capture complex usermode initialization in the guest. In this case, the start time of the JVM and our application adds significantly to the cold-start time of the virtual machine. The combined initialization time is significant enough to make deploying such an application on a serverless platform latency-prohibitive. We aim to demonstrate that this cold-start time, in addition to the standard boot time, is significantly improved by HyperFork, allowing the high-latency Spark framework to be used.

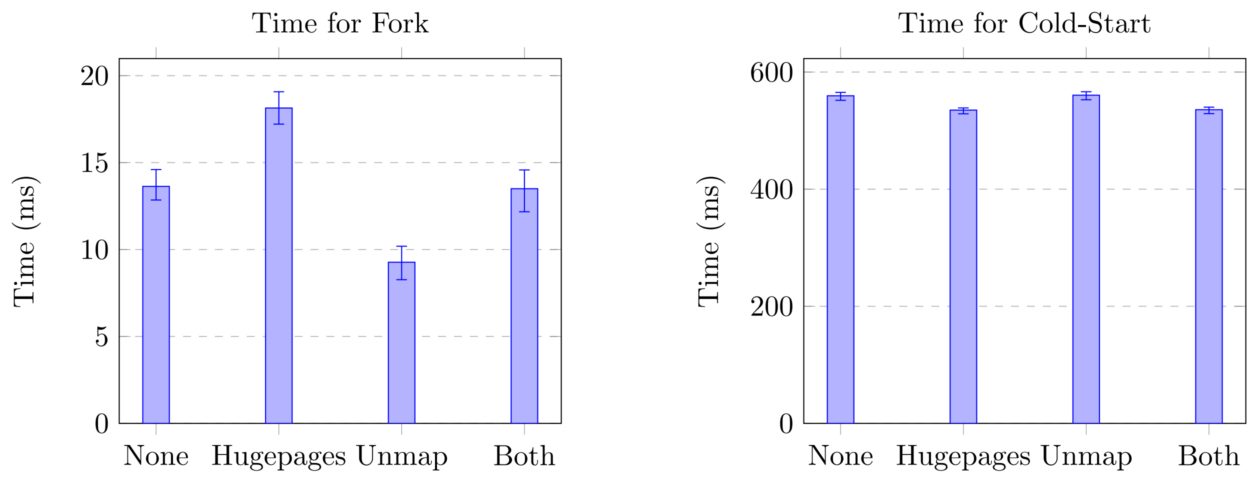

Figure 7: Time for virtual machine fork vs. cold-start, with 1GB of memory and various optimizations

Compile benchmark. The compile benchmark tests HyperFork's performance for a virtual machine compiling a small C program. The virtual machine boots a Linux guest, requests a flash-clone, then compiles a small C program using `gcc`. This benchmark uses a combination of CPU, memory, and disk resources in the guest to assess any performance degradation experienced by flash-cloned guests.

Throughput tests. For the Java and compile benchmarks, we further assess HyperFork to estimate its throughput in a serverless deployment. To do so, we modify HyperFork to suspend the parent virtual machine when it requests a flash-clone. It then flash-clones one virtual machine per CPU, creating a new one from the suspended parent whenever a child virtual machine completes. The rate at which jobs are completed is measured. We compare this to the same scheme using cold-started virtual machines.

4.3 Results

Fork performance. Figure 7 displays the performance of the flash-clone operation compared to a fresh boot of the same image. Error bars indicate the 2.5- and 97.5-percentiles. We compare the two methods of generating a new virtual machine with and without our memory unmapping optimization (§3.3), and with and without hugepages. The virtual machine was booted with 1GB of RAM and simply performed a fork operation before exiting. With the unmapping optimization, HyperFork performed 60x faster than a naive boot. New 1GB virtual machines were spawned in 9.27 ms on average, compared to 560 ms from cold-starting. Note that in this setting, we observed performance degradation when hugepages were enabled. As we will see, the performance effect of hugepages is highly dependent on the amount of memory allocated to the VM. For future

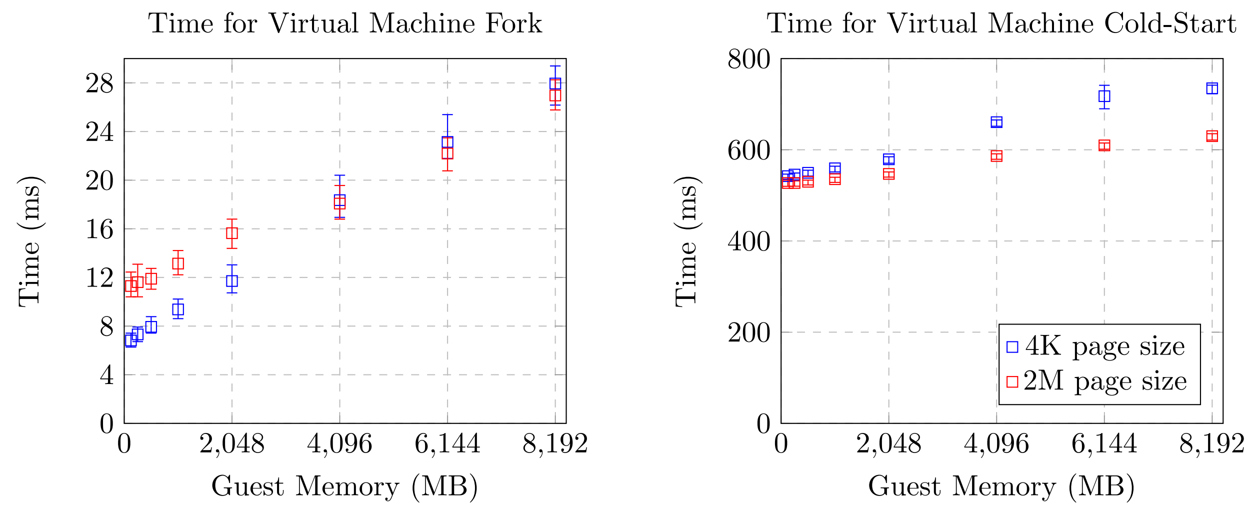

Figure 8: Time for virtual machine fork vs. cold-start, with unmap optimization and both page sizes

benchmarks, the unmapping optimization is left enabled.

Figure 8 compares the performance of the flash-clone operation with boot times for various amounts of virtual machine memory. Again, error bars denote 2.5- and 97.5-percentiles. Across the board, HyperFork performs more than 20x faster than a naive boot, both with and without hugepages enabled. The time required to perform a flash-clone grows roughly linearly with the amount of memory allocated to the virtual machine. This is roughly what we expect, given that most of the flash-clone operation is spent in pagetable duplication routines. The number of entries to copy will grow linearly in the amount of memory.

Furthermore, we note that for virtual machines with small amounts of memory, HyperFork performs better with normally sized pages. This is likely due to the overhead duplicating pages during and after the fork operation. Specifically, after the flash-clone, both the parent and child will share the pages backing their virtual memory, duplicating them when a write is performed. For regular pages, this requires 4KB to be duplicated each time a fault occurs. For hugepages, 2MB must be duplicated, even if only a single byte is changed. For sufficiently small images, the overhead of these CoW duplications dominates and makes hugepages less efficient.

As the memory allocated to the guest increases, hugepages become more performant, beating out regular sized pages after the memory size is 4GB or more.

Memory performance. Results from our memory benchmark are displayed in Figure 9. The "Unbacked" pass is the first pass, in which guest physical pages are not yet backed in the host. Each access to an unbacked page triggers a page fault to allocate a page. We note that hugepages significantly decrease the overhead of unbacked pages, as each page fault allocates a 2MB region

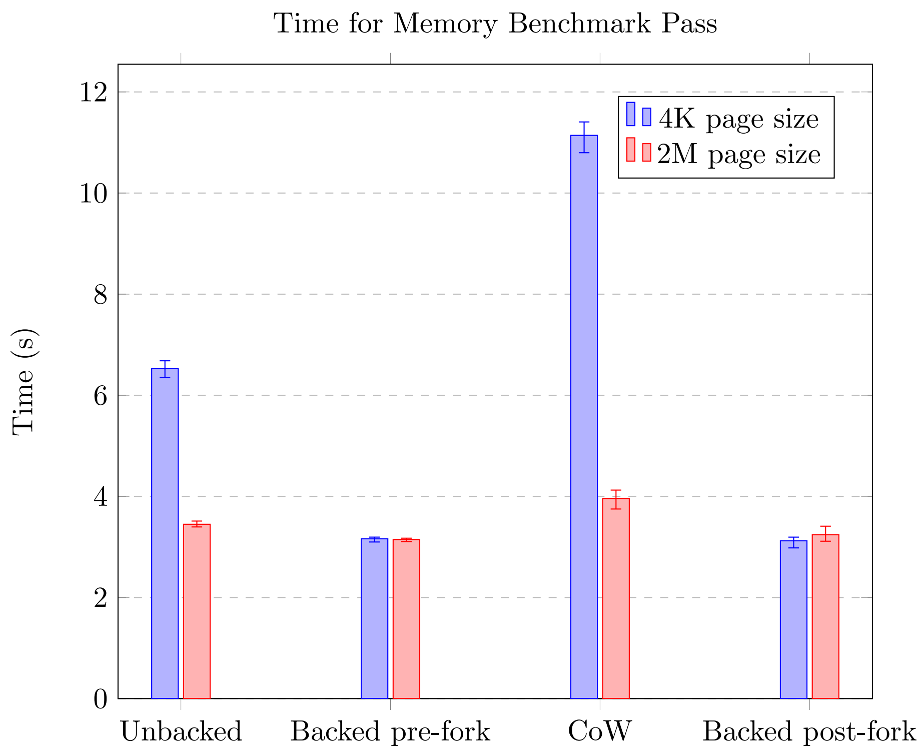

Figure 9: Pass times for the memory benchmark

instead of a 4KB one. This results in fewer page faults and increased performance on the benchmark.

Contrast the unbacked data with the third pass ("CoW"). In this pass, all pages are shared between the parent and child VM. Thus every write to a page will trigger a copy-on-write page fault in the host, which must duplicate the page and update both pagetables. We observe that copy-on-write produces a 70% performance degradation compared to unbacked pages with a regular page size. This comparison is relevant, as all guests produced through flash-cloning will resume with copy-on-write pages, whereas guests produced through naive booting will have only backed and unbacked pages. Therefore this indicates the worst-case memory performance penalty from sharing memory between guests. However, note that this overhead shrinks to 14.8% when hugepages are used. Therefore, 2MB pages may be more appropriate to maintain performance in memory-bound guests under HyperFork.

Finally, we note that the two backed passes have nearly equivalent performance. This aligns with intuition, as after pages are backed or duplicated, no further page faults to the host will be required and we expect no further performance degradation in a flash-cloned guest.

Java and GCC performance. We next measured the amount of time that the Java and GCC jobs took to complete after the fork (or boot and initialization in the case of a cold-start) was complete. Results are displayed in Figure 10. The GCC benchmark displays negligible performance

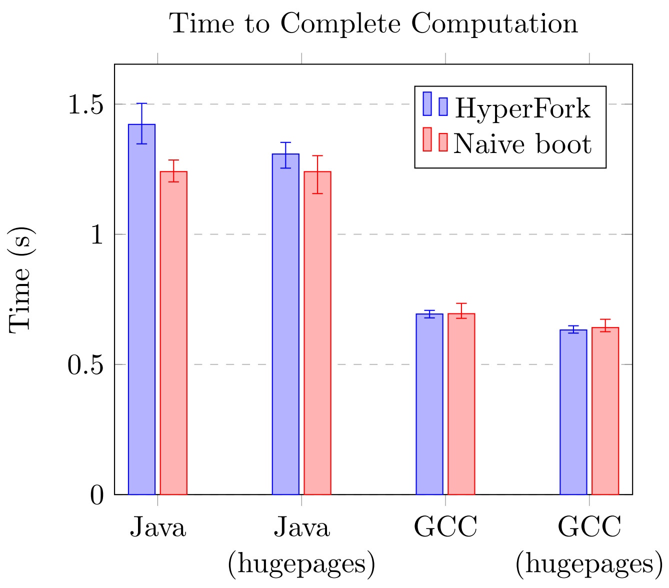

Figure 10: Time required for computation in forked and cold-started virtual machines, excluding boot/initialization time

differences between the two virtual machine creation methods.

The Java benchmark, on the other hand, exhibits a 15% performance penalty when produced by forking instead of cold-starting. We suspect that this is caused by the overhead of duplicating shared pages in the JVM's large memory footprint. This overhead is reduced to 5.5% when hugepages are used instead. These results indicate that HyperFork's impact on the performance of jobs run after the flash-clone operation is small. In memory-intensive workloads, hugepages can decrease this overhead further.

Throughput. Our throughput benchmarks demonstrated the HyperFork significantly increases throughput over booting fresh virtual machines for each job. Figure 11 displays these results for the GCC and Java benchmarks. The Java benchmark, which required nearly 6 seconds to boot and initialize the JVM and required Spark libraries, exhibited a 502% increase in throughput when using HyperFork. Flash-cloning virtual machines eliminated the overhead of loading the large Spark library into memory and preparing the job for execution. HyperFork also reduced total system CPU time for the benchmark by 84%.

The GCC benchmark displayed a 46% increase in throughput under HyperFork. Because this benchmark does not require any initialization beyond booting, it does not benefit as much as the Java benchmark from flash-cloning. However, this is still a significant performance increase and is matched by a 38% reduction in system CPU time.

For each simulated request in the throughput test, the time required to complete the job was recorded. Using these completion times, we can compare the latency of a serverless deployment at

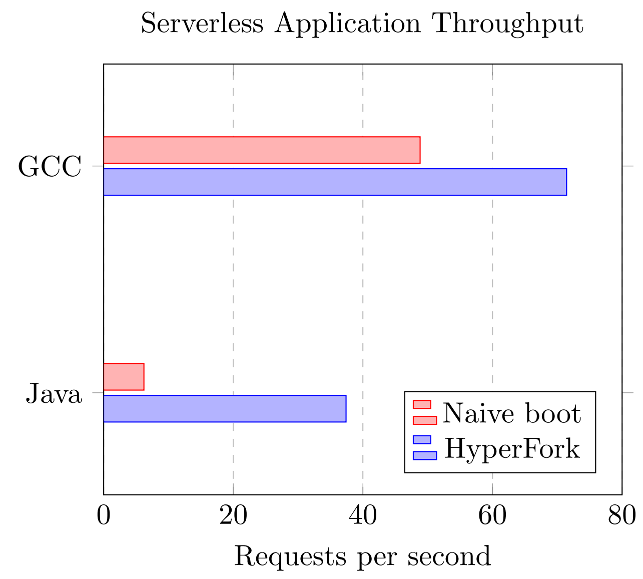

Figure 11: Throughput of Java and GCC serverless benchmarks, HyperFork vs. naive boot

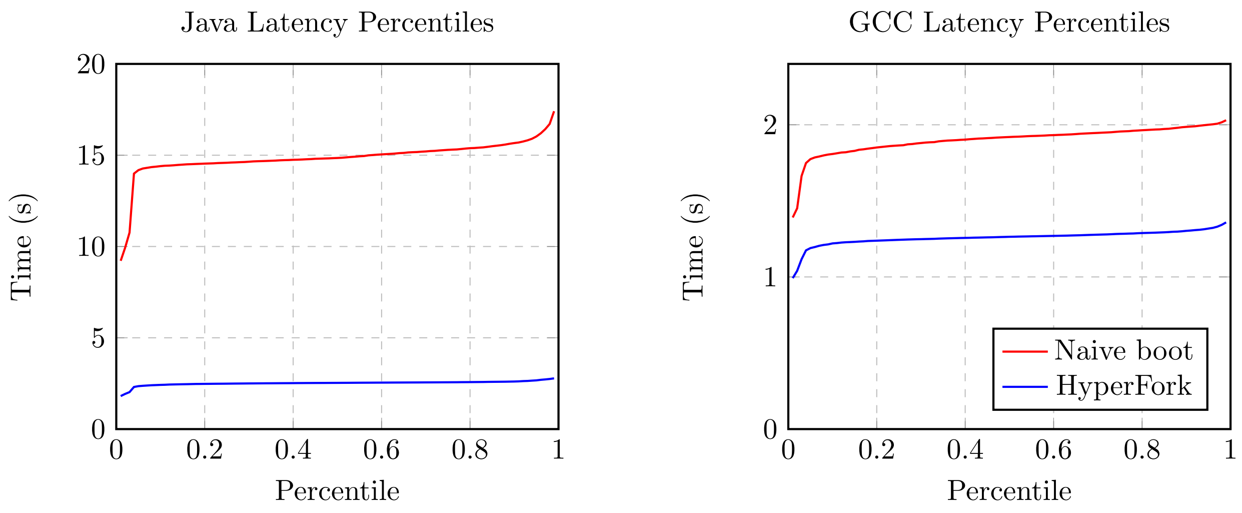

Figure 12: Latency of requests in throughput simulation

capacity using HyperFork and a naive virtual machine boot. Figure 12 displays these percentiles. We see that latency has decreased significantly for all percentiles on both benchmarks, with more drastic results on the Java benchmark due to its long initialization time. HyperFork decreases p95 response times by 83.3% for Java and 34.1% for GCC.

5 Discussion

Our results have demonstrated that HyperFork is a highly performant alternative to cold-starts of virtual machines, significantly decreasing start time while increasing throughput. Our start times beat out Firecracker's reported 125 ms start times by a significant margin, even for very large memory sizes [32]. Furthermore, the decreased start latency made it feasible to deploy a serverless application based on the latency-prohibitive Spark framework, improving the resulting median job latency by a factor of 5.8. Therefore HyperFork enables a large class of heavyweight frameworks to be used in serverless jobs without sacrificing latency.

We now assess some security and correctness concerns of our approach, and conclude by discussing future directions for research.

5.1 Security concerns

Despite HyperFork's significant increases in performance over a naive cold-start method, maintaining security in the forked guests remains a challenge. We discuss a series of potential security weaknesses and approaches for mitigating them.

Materialized randomness. While the PRNG in the guest kernel can easily be reseeded to ensure that guests have independent entropy pools, addressing materialized randomness is much more difficult. As discussed before, there are no easy mechanisms to rerandomize the layout of the kernel and update KASLR. Because a serverless framework often allows an end-user to spawn as many instances of the job as desired, all of these instances may have the same kernel address space layout. Should a kernel exploit exist, this opens a pathway to defeat KASLR by brute force, similar to approaches applied to ASLR [33].

ASLR itself may also be weakened by the approach taken in HyperFork, as processes started before the fork occurs will have identical layouts in all child virtual machines. Again, this may allow ASLR to be defeated through brute force by triggering a large number of events and attempting to exploit each new virtual machine.

Materialized randomness in userland applications must also be addressed. Consider, for example, a PRNG used in an application which seeds itself with the kernel PRNG when the application starts. The PRNG never requests new entropy from the kernel after this initialization. Should this seeding occur before the flash-clone, application PRNG states will remain identical between guests even once the kernel entropy pool has been reseeded. Thus, maintaining the security of userland PRNGs requires applications and frameworks to reseed their entropy pools after the fork has occurred.

To more generally mitigate these problems, discarding suspended virtual machines after a certain number of forks will help to avoid brute force attacks with a small performance penalty. For example, a certain virtual machine may only be permitted to fork 1000 times before it is freshly booted. Therefore an adversary obtains at most 1000 attempts to brute force the materialized randomness. This approach rectifies many of the issues with KASLR and ASLR, but may not resolve the use of materialized userland randomness in initializing other cryptographic primitives.

Timing side-channels between guests. Because guests produced through flash-cloning have different memory performance characteristics than those produced through cold-starts, timing side-channels may exist to extract information about the state of other guests. Specifically, after two guests are forked from the parent, all three processes share the same pages with copy-on-write semantics. When the page is written in one of the guests, a page fault is triggered. This page fault is significantly slower than a standard memory access and somewhat slower than a page fault due to an unbacked page.

A guest could conceivably modify a series of pages and measure the amount of time it takes for the process to complete. This timing information can be used to estimate how many other guests have those pages mapped, as we expect page fault time to increase with the number of processes sharing the page. This leaks information about the state and behavior of other guests, violating the isolation boundary between guests.

However, we note that these types of timing attacks are not unique to HyperFork. Delimitrou and Kozyrakis were able to stage a similar attack across independently started VMs, using timing information to deduce the types of jobs running on the system [34]. In the case of HyperFork, however, we have much more detailed knowledge about the initial state and context of the guests and may be able to extract more specific or sensitive information. For highly sensitive tasks, these attacks may be cause to require jobs from different tenants to originate from different parent virtual machines.

Incorrect isolation in the VMM. Because multiple guests originate from the same parent virtual machine, some resources remain shared between the address spaces. As discussed previously, much of the HyperFork implementation simply recreates these resources in child processes to separate them from the parent. However, note that if some resources are incorrectly duplicated, state may remain shared between the parent and child, potentially resulting in guests violating isolation. Over the course of our development, we experienced deadlocks arising from incorrectly duplicated condition variables, pseudoterminal controls, and shared `eventfd`s.

While each of these problems were corrected as they were encountered, they highlight that shared or inconsistent state can arise in unexpected places due to `fork`'s interactions with threads and file descriptors. We have not encountered any deadlocks or isolation violations since these problems were resolved, but guaranteeing that the guest process is fully independent is quite difficult to do without examining file descriptor and kernel state.

5.2 Future work

We now consider additional work that would be required or helpful for HyperFork to be deployed as the hypervisor for a multitenant serverless deployment running a large set of jobs.

Additional hardware support. In many serverless deployments, such as Amazon Lambda, virtual machines communicate with their hypervisor via a network interface. Once the virtual machine has started, the hypervisor communicates its task or context via the network interface, and the virtual machine eventually responds with the result of its computation. We have not yet ported the `kvmtool` network device to support flash-cloning. This likely involves simple userland state duplication and updating the network configuration in the guest. Alternatively, some form of NAT could be performed in the hypervisor to avoid changing the network context in the guest.

Virtual machine trees. As a standard serverless deployment will be running a set of unique virtual machine images from multiple tenants, the current implementation of HyperFork would require each image to be booted independently. This imposes significant overhead and would likely require the job dispatch system to dispatch identical jobs to the same server in order to benefit from HyperFork. Note that many serverless deployments already do this in order to minimize the number of times virtual machine images must be retrieved from disk.

However, even though the content of each virtual machine is unique to each serverless application, they do share some commonalities. For example, a large set of images may use the same version of the Linux kernel, which must be booted for each and every application. We can consider some of the process of booting a virtual machine to be common across applications and perform the operation only once. When the applications begin to diverge (e.g. by using a different disk image once the kernel has loaded), we can perform a flash-clone, attach the disk, and allow the virtual machines to proceed independently. This would enable us to avoid cold boots for all applications that use a common kernel version.

Note that a similar strategy may be applied to state initialized after booting has completed. Consider, for example, a set of applications that each require a local database to be started. HyperFork could simply boot an image with just the database, initialize it, and then flash-clone. In each of the children, a disk containing the specific application can then be loaded. While this approach requires application virtual machines to be stored in a different format, it allows the initialization time of the database to be avoided.

In general, for a serverless deployment hosting applications with specific runtimes, commonalities can be factored out and prepared in a single virtual machine execution. The resulting virtual machine is then forked, the next commonality is attached as a new disk, and the machines are resumed. This process continues until each of the runtimes is prepared, except for the application running on that specific runtime. In this way, a tree of virtual machine states is constructed by invoking flash-clone when a set of runtimes diverge. In the end, we are left with a set of suspended virtual machines. We expect this method will be able to accommodate a large number of runtimes, as unmodified memory will be shared between the various virtual machines via copy-on-write. Therefore, each runtime only needs to allocate memory for the pages which differentiate it from its parent. We leave evaluating the performance and resource consumption of this approach to future work.

Integration with a serverless dispatcher. To assess the performance of HyperFork in a more realistic serverless environment, it would be helpful to integrate it with a commonly used serverless dispatch platform such as OpenWhisk. This would allow us to more accurately measure the latency and throughput characteristics when executing heterogeneous jobs. It would also provide a platform with which to directly compare HyperFork to Firecracker and similar work.

6 Conclusion

We have presented HyperFork, an extension to `kvmtool` that enables flash-cloning of virtual machines. Our evaluation demonstrated that flash-cloning improves VM spawn times by a factor of up to 60, while offering significant improvements in serverless throughput and latency. We further identified means of improving performance by avoiding slow paths in the KVM memory management implementation and utilizing hardware hugepages where appropriate. Our benchmarks demonstrated that HyperFork makes a Spark application that was previously infeasible to run on a serverless platform both responsive and efficient. Together, these results indicate that flash-cloning is a powerful means of reducing latency and enabling a wider class of applications to be deployed on serverless platforms.

Acknowledgements

I would like to thank my thesis advisor Eddie Kohler for his support, guidance, and encouragement throughout the entirety of this project, as well as my time at Harvard. I would further like to thank and recognize Mark Wilkening and David Gardner for their contributions to the original implementation and report on HyperFork.

References

[1] Firecracker frequently asked questions. `https://github.com/firecracker-microvm/firecracker/blob/master/FAQ.md`, accessed 2019.

[2] Fabrice Bellard. Qemu, a fast and portable dynamic translator. pages 41–46, 01 2005.

[3] Andrew Whitaker, Marianne Shaw, Steven D Gribble, et al. Denali: Lightweight virtual machines for distributed and networked applications. Technical report, Technical Report 02-02-01, University of Washington, 2002.

[4] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 164–177, New York, NY, USA, 2003. ACM.

[5] Rusty Russell. Virtio: Towards a de-facto standard for virtual i/o devices. *SIGOPS Oper. Syst. Rev.*, 42(5):95–103, July 2008.

[6] Jack Lo. Vmware and cpu virtualization technology. `http://courses.cs.vt.edu/~cs5204/fall07-kafura/Papers/Virtualization/VMWare-Technology.pdf`, accessed 2020.

[7] Julie Brodeur. Binary translation deprecation. `https://docs.vmware.com/en/vSphere/6.7/solutions/vSphere-6.7.2cd6d2a77980cc623caa6062f3c89362/GUID-B16CF6F8A1D2742AC9A9546E1F533384.html`, accessed 2020.

[8] Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, pages 2–13, New York, NY, USA, 2006. ACM.

[9] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. Kvm: The linux virtual machine monitor. *Proceedings Linux Symposium*, 15, 01 2007.

[10] kvmtool. `https://github.com/clearlinux/kvmtool`, accessed 2019.

[11] The firecracker virtual machine monitor. `https://lwn.net/Articles/775736/`, accessed 2019.

[12] Anil Madhavapeddy and David J. Scott. Unikernels: Rise of the virtual library operating system. *Queue*, 11(11):30:30–30:44, December 2013.

[13] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Denali: A scalable isolation kernel. In *Proceedings of the 10th Workshop on ACM SIGOPS European Workshop*, EW 10, pages 10–15, New York, NY, USA, 2002. ACM.

[14] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. Peeking behind the curtains of serverless platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 133–146, Boston, MA, July 2018. USENIX Association.

[15] Aws lambda announces provisioned concurrency. `https://aws.amazon.com/about-aws/whats-new/2019/12/aws-lambda-announces-provisioned-concurrency/`, December 2019.

[16] `https://www.docker.com/`, accessed 2019.

[17] `https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt`, accessed 2019.

[18] Cve-2019-5736. `https://nvd.nist.gov/vuln/detail/CVE-2019-5736`, accessed 2019.

[19] Michael Vrable, Justin Ma, Jay Chen, David Moore, Erik Vandekieft, Alex C. Snoeren, Geoffrey M. Voelker, and Stefan Savage. Scalability, fidelity, and containment in the potemkin virtual honeyfarm. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP '05, pages 148–162, New York, NY, USA, 2005. ACM.

[20] Michael R. Hines and Kartik Gopalan. Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '09, pages 51–60, New York, NY, USA, 2009. ACM.

[21] Horacio Andrés Lagar-Cavilla, Joseph Andrew Whitney, Adin Matthew Scannell, Philip Patchin, Stephen M. Rumble, Eyal de Lara, Michael Brudno, and Mahadev Satyanarayanan. Snowflock: Rapid virtual machine cloning for cloud computing. In *Proceedings of the 4th ACM European Conference on Computer Systems*, EuroSys '09, pages 1–12, New York, NY, USA, 2009. ACM.

[22] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual*, October 2019. Order number 325462-071US.

[23] Andrew Baumann, Jonathan Appavoo, Orran Krieger, and Timothy Roscoe. A fork() in the road. In *17th Workshop on Hot Topics in Operating Systems*. ACM, May 2019.

[24] pthread_mutex_lock(3). `https://linux.die.net/man/3/pthread_mutex_lock`, accessed 2020.

[25] Stephan Müller. Linux random number generator - a new approach. `https://www.chronox.de/lrng/doc/lrng.pdf`, 2020.

[26] Jake Edge. Kernel address space layout randomization. https://lwn.net/Articles/569635/, 2013.

[27] The x86 kvm shadow mmu. https://www.kernel.org/doc/Documentation/virtual/kvm/mmu.txt, accessed 2019.

[28] Takuya Yoshikawa. How to use kvm's reverse mappings to improve scalability. https://www.linux-kvm.org/images/f/fd/2012-forum-yoshikawa.pdf, accessed 2019.

[29] https://www.kernel.org/doc/Documentation/vm/hugetlbpage.txt, accessed 2019.

[30] busybox. https://hub.docker.com/_/busybox, accessed 2020.

[31] busybox(1). https://linux.die.net/man/1/busybox, accessed 2020.

[32] Announcing the firecracker open source technology: Secure and fast microvm for serverless computing. https://aws.amazon.com/blogs/opensource/firecracker-open-source-secure-fast-microvm-serverless/, accessed 2020.

[33] Tilo Müller. Aslr smack & laugh reference. In *Seminar on Advanced Exploitation Techniques*, 2008.

[34] Christina Delimitrou and Christos Kozyrakis. Bolt: I know what you did last summer... in the cloud. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, page 599–613, New York, NY, USA, 2017. Association for Computing Machinery.