HyperFork: Virtual Machine Flash Cloning for Serverless Computing

Michael Colavita David Gardner Mark Wilkening
Harvard University

ABSTRACT

We present an abstract!

1. INTRODUCTION & MOTIVATION

Serverless Computing. Serverless computing, also known as *Functions-as-a-Service* (FaaS), has become an increasingly important and desirable platform in the cloud ecosystem. Stemming from the grand vision of computation as a utility, serverless computing offers users the ability to run application code directly on a black-box infrastructure. In comparison to current cloud computing platforms, serverless users (1) no longer have to manage virtual machine environments, (2) are billed only for application computation performed in response to requests, and (3) function instances are auto-scaled to handle dynamically changing request rates. Options are available from all of the major cloud players, including Amazon Lambda, Azure Functions, and Google Cloud Functions. Several significant online companies have implemented parts of their services on serverless platforms, notably the news site The Guardian.[1].

Serverless Infrastructure. The typical implementation of a serverless computing platform places user-submitted functions onto dynamically created Virtual Machines (VMs). These functions are intended to be light-weight stateless single-process programs. Because of the relatively short run-times of serverless functions, a critical performance constraint in serverless infrastructure implementations is the scheduling latency of functions – mainly comprising of the creation time for instance VMs. A standard optimization employed for this start-up time is to keep instance VMs running for a period of time, and schedule function requests onto existing VMs. This leads to a number of other properties, including limits on function run-time and the potential for functions to be terminated at any time. This is necessary for the scheduler to auto-scale instances efficiently. Another property is the clear split in individual function start-up latencies between *warm-starts*, where a function is scheduled to an already running instance, and *cold-starts*, where a new VM must be created for the function. Functions from different users are usually not placed in the same VM for security and isolation reasons, but one VM can host several instances of one user's function.

[1]https://aws.amazon.com/solutions/case-studies/the-guardian/

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

The Problem with Serverless: Cold-start. A central promise of serverless computing services is rapid scalability. Meeting this demand at scale requires that new function instances can be started very quickly to service incoming requests. This is easy when there exist currently running warm instances but more difficult when a new cold instance must be started. A major bottleneck in achieving low latency instance start-up and fast auto-scaling is the cold-start latency [?]. For cold-start, the major bottleneck is the creation of a new VM. After scheduling and provisioning, when a VM is created the host Hypervisor/Virtual Machine Monitor (VMM) must initialize virtual resources including CPUs, memory, and other devices, then the guest kernel and file system must be loaded from disk and initialized in guest memory, then the guest kernel is booted, and finally the function runtime/process is started and the request is handled.

Flash Cloning. The most significant parts of VM creation are (1) copying the kernel and file system into memory, (2) booting the guest kernel, and (3) loading potentially large libraries/runtimes. Management operations within the VMM are relatively cheap compared to the start-up within a guest and the copying of guest memory. Optimizing these steps could dramatically reduce the startup latency of new serverless functions. One technique to do this is to employ *flash cloning.* Instead of loading VM images from disk and booting a kernel, we propose cloning existing reference VMs in memory. Additionally, we propose a copy-on-write mechanic to reduce both the copy time overhead and the memory pressure of packing many VMs onto one host. This method can be compared to the Unix fork abstraction.

> To create an isolated virtual machine, rather than re-create the entire world, we should only re-create the necessarily distinct VMM components and clone identical guest memory and execution context from ready-to-go reference VMs.

From this insight we present *HyperFork*, a KVM-based VM cloning implementation for the context of serverless computing.

Figure 1: KVM Software Architecture

Contributions. Our contributions are as follows:

- A KVM based virtual machine cloning implementation which outperforms standard VM creation latencies by up to AMAZING%
- A thorough discussion of the trade-offs related to a number of potential implementations for VM creation in the context of serverless computing.
- A thorough analysis of the performance of Hyperfork for both VM creation latency as well as VM co-location density and performance degredation with respect to copy-on-write memory sharing.

The rest of the paper is organized as follows. Section 2 provides an overview of KVM and the hypervisor *kvmtool*, and then describes the implementation details of *HyperFork*. Section 3 describes our experimental evaluation. Section 4 presents a background and review of related work.

2. HYPERFORK ARCHITECTURE

In this section we describe our HyperFork architecture and implementation, including the overall platform architecture, the relevant implemetation details of KVM and kvmtool, as well as our design decisions regarding the flash-cloning operation.

2.1 Platform Architecture

Describe Xen vs KVM. kvmtool vs. Firecracker. Amazon Linux vs our small kernels. This is where many of our foundational design decisions should be justified. The following sections describe our implementation decisions.

2.2 KVM Overview

A full KVM system contains a number of major components within a Linux host machine, as depicted in Figure 1. The KVM kernel module tracks and maintains most of the sensitive virtualized machine state. Each guest is isolated within a linux process, which contains both VMM management components as well as the running guest. Within the VMM, management components withing the userspace guest processes communicate with the KVM kernel module via a set of IOCTLs. Outside of the guest processes, the VMM contains CLI based management utility programs for administrators which use IPCs to communicate with the guest process VMM components. For our implementation, the userspace VMM components are implemented in kvmtool, however these same fundamental components will exist in other KVM VMM implementations such as Firecracker [**?**].

2.2.1 *kvmtool*

kvmtool [**?**] is a VMM implementation for KVM with the minimal functionality required to boot a fully functional linux kernel with very basic virtualization devices. The 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 [**?**], which supports a wide range of legacy devices and guest configurations.

We selected kvmtool as our userspace VMM as it offers a very similar set of functionality to Firecracker, the VMM used to power Amazon Lambda. (TODO: contrast firecracker and kvmtool) We chose kvmtool over Firecracker however because we found that Firecracker was more difficult to work with due to its Rust codebase and containerization schemes. kvmtool provides a minimal platform on which to test HyperFork when applied to Linux guests and closely approximates the VMM of an industrial serverless platform.

To virtualize efficiently, kvmtool makes use of a number of threads for managing vCPUs and emulated devices. Userspace bookkeeping data structures hold file descriptors which point to the sensitive VMM state maintained within 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. It then creates several worker threads to handle arbitrary jobs that may arise from the virtio devices. These tasks include processing work items from virtio queues and updating the console. In its default configuration, kvmtool allocates one worker thread for each CPU on the host machine. As we are virtualizing machines that are much smaller than the host machine, we limited kvmtool to one worker thread per VM. In addition to device threads, kvmtool also creates a thread to manage the virtual machines through IPC calls. This allows administrators to start, pause, stop, and debug virtual machines using a simple command line interface. Finally, kvmtool creates one thread per vCPU that proceeds in a loop, invoking the `KVM_RUN` ioctl, then handling any IO requests or interrupts that may arise. Together, this set of threads, enable efficient virtualization of the guest and its devices.

2.3 HyperFork Implementation

For our KVM based virtualization platform, flash-cloning requires duplicating a number of important components. We would like to make heavy use of the linux fork operation to duplicate guest memory with copy-on-write functionality, as well as duplicating much of the userspace VMM management state. Fork alone however still leaves kernelspace management state unduplicated, cannot duplicate all userspace VMM threads, and leaves userspace bookkeeping file descriptors pointing to parent kernelspace management state. Duplicating kernel state can either be performed within the kernel or entirely in userspace by extracting and saving guest specific state in userspace before forking. We predict a kernel implementation may have slightly higher performance, but for simplicity we implement our flash-cloning entirely in userspace within kvmtool. We also add an ad-hoc guest-to-host communication channel, which is not supported by default within kvmtool, to enable our experimental evaluations.

2.3.1 *Flash-Cloning Support in kvmtool*

Our userspace HyperFork implementation for kvmtool proceeds in a series of phases. The phases include a triggering event, pre-fork extraction and saving of kernel state, forking, and post-forking reconstruction of VMM management state in the child.

First, the fork is triggered, either by an administrator invoking the `FORK` IPC via the command line interface, or by the guest sending a signal to the host indicating that it is ready to fork. In either case, the IPC thread receives this signal, pauses the virtual machine, and calls the pre-fork routines.

Because KVM state becomes inaccessible in the child process after the VMM has forked, all kernel state that must be restored in the child needs to be recorded by the parent before forking. Alternatively, this could be implemented by IPC between the parent and child, in which the state is sent after the fork is complete. We have adopted the former approach. The pre-fork routine thus saves all individual vCPU state for all vCPUs (registers, interrupt configuration), global vCPU state (interrupt configuration, clock), and then locks all mutexes that must survive in the guest. Note that it is not necessary for the pre-fork routine to save the memory of the virtual machine. As the memory is mapped in the VMM process, it is unaffected by the fork system call and remains accessible. It will, however, need to be remapped in the guest following the fork.

Once the pre-fork routine is complete, kvmtool performs a fork. In the parent, all of the locks acquired by the pre-fork routines are released and execution proceeds. In the child, the post-fork routine is invoked, performing the following to reconstruct necessary VMM state:

1. Acquires new file descriptors for the KVM device and virtual machine
2. Creates new file descriptors for the vCPUs
3. Restores individual and global vCPU state[2]
4. Replaces all eventfds used for signalling
5. Creates new threads to handle devices and the execution of each vCPU
6. Releases all mutexes locked in the pre-fork routine, and replaces all condition variables[3]
7. Attaches the terminal device to a new pseudo-terminal, or detaches it to accept no further input[4]

Once the post-fork routine is complete, the vCPUs begin executing in the child and the flash-clone operation is complete. We can also note that the original thread in the parent which performed the forking operation was the IPC handling thread. In the child, this thread creates a new IPC handling thread and then assumes the role of the main kvmtool run thread, waiting on the guest vCPU0 to exit and then terminating the VM gracefully.

2.3.2 Guest-to-Host Signalling

For guests with more complex forking behavior, the guest may need to inform the host when it is ready to fork. For example, a virtual machine running a python program may chose to fork on boot, after python has started, after modules are loaded, or after further program initialization has completed. As the guest's userspace state is very difficult to detect from the VMM, we implement a rudimentary system for guest-to-host signalling.

The guest-to-host signal consists of sending a message over one of the processor's ports. This allows for a simple and very efficient way to send short messages to the host, without requiring any modifications to the guest kernel. This functionality is accessible from userspace through the C standard library. We define one message to indicate that the guest is ready to fork, and one message to indicate that the guest has completed its task. To support our evaluations, we include in the guest images a `fork` and `done` executable that signal the two events which can be easily executed by guest benchmarks. Further messages could easily be defined for a more complex deployment.

3. EXPERIMENTAL EVALUATION

This is the experiments section First summarize the findings in an implicit way. For example: "In this section, we demonstrate that System X achieves Z, Y, K" where Y, Z, K are our main contributions in the paper as well.

Experimental Setup.

We perform all tests on [SOME MACHINE], with [SOME SPECS]. The file system image used for all tests is a minimal Linux setup generated using the buildroot utility.[5] This specially generated image is not any particular distribution, but contains only the bare minimum of utilities and software needed for our benchmarks. In typical use, the guest images used for FaaS platforms may be more fleshed-out, but such single purpose images have no need for many of the qualities of a more feature-rich distribution. Since FaaS uses very short-lived instances, there is no need for package managers, update utilities, or other maintenance software.

Using a minimal image also presents cold-boot times in the best possible environment, as a heavier image would take longer to boot. We therefore show that our implementation offers substantial start time improvements over even the best case scenario of cold-booting.

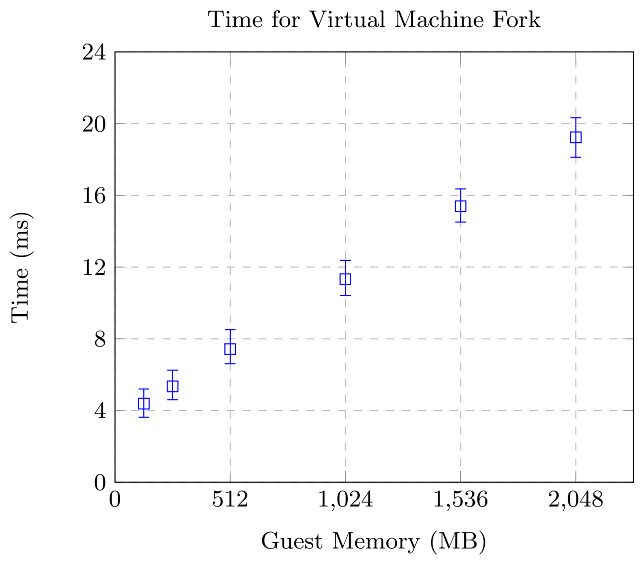
3.1 Benchmarks

[2]In the process of restoring this state, we encountered a bug in how KVM handles setting the control registers when they change whether the guest is in long mode. We intend to investigate this further and report it if necessary.

[3]Condition variables must be replaced, as in many pthread implementations they contain an internal mutex that cannot be locked in the pre-fork routine. If this mutex is locked by another thread when the IPC thread performs the fork, the mutex will be permanently locked in the child process.

[4]Due to a bug in kvmtool, operating with a pseudoterminal with no slave is not supported.

[5]buildroot.org



Fork Time. We run several experiments to test the time for a VM to fork. These tests start a virtual machine, allocate some memory, and then fork. The forking process marks a timestamp when its fork() call returns, and the child marks a timestamp once it finishes restoring KVM state.

Even though memory is not copied to the child process unless it is written, it still takes time to walk through the parent page table and mark shared memory as read-only, and generate page tables for the child process. For this reason we expect to see fork times scale roughly linearly with the amount of memory allocated pre-fork. Allocating more memory simulates a VM with more programs and libraries loaded. [FIGURE: fork time in parent and child vs amount of memory allocated pre-fork]

Copy-on-Write Test. Though cloning VMs using Fork can decrease VM start times by orders of magnitude because of shared memory, performing Copy-on-Write on those shared regions has the potential to reduce performance. We implement a benchmark to isolate the performance degradation caused by Copy-on-Write operations after cloning.

This program begins by allocating p pages of memory, and writes b bytes of random data to each one. It then signals to the host, which forks the VM n times. In each child the program then writes more random data to each of those pages.

Each of those post-fork writes will trap into the host kernel and induce a copy. We compare the average time to completion of child VMs to the time it takes a VM that has not forked to perform the same writes.

Real-world Conditions. In the lifecycle of a typical Function-as-a-Service unit, a VM is started, code and data are copied on, and the user function is run. These services mostly use runtimes like NodeJS or Python. To simulate this sort of workload, we use pillow-perf, a Python image processing benchmark. This captures the typical scenario of a function which loads an interpreter, loads some external libraries, and performs some CPU-bound computation.

We compare the total time of starting n separate VMs, and letting each run the benchmark to completion, with the total time of forking n VMs from one reference image and letting them run to completion. [TODO: fork at different times] The CPU performance of forked VMs should not differ from separately booted VMs. Using a CPU-bound benchmark like pillow-perf verifies this assumption.

3.2 Results

Explaining Each Figure. We first put the most important figures, i.e., the ones that have the most important results in terms of the contributions. Each figure should be a single message. Each figure comes with two paragraphs. One paragraph for the setup and one paragraph for the discussion. The set-up paragraph should start by saying "In this experiment we show that... We setup this experiment as follows...".

The result paragraph explains in detail why we see what we see. Explanations should be based on facts and logic. Numbers that back up the explanations should be provided whenever possible. The paragraph should finish by repeating the main message again.

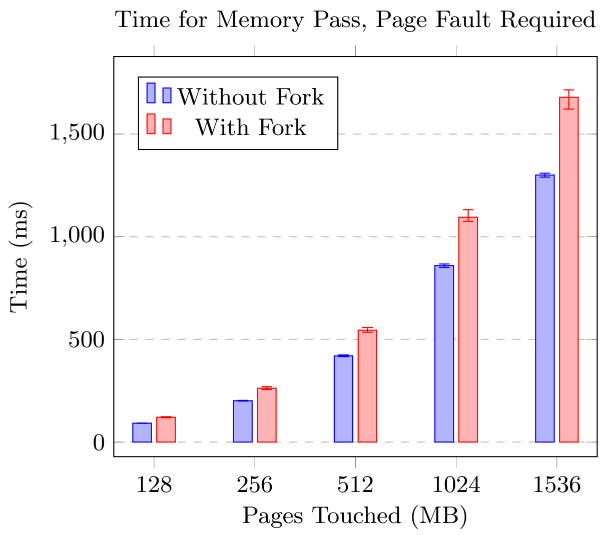
4. RELATED WORK OR BACKGROUND

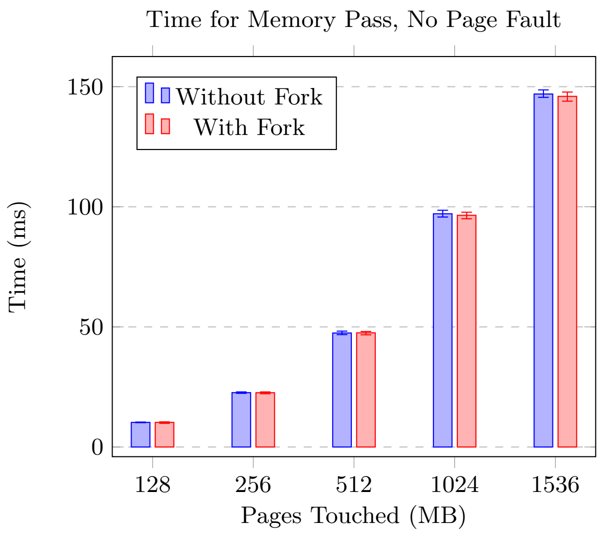
Our work in this paper draws on concepts from several lines of past research.

Virtualization Technology. Machine virtualization technology is a complex and diverse space. Classical Hypervisors/Virtual Machine Monitors (VMMs) relied on the fundamental primitive of *Trap-and-Emulate*, where sensitive instructions in the guest would be trapped by the hardware, and emulated safely within the VMM using shadow structures for privileged state [?]. In x86 however, not all sensitive instructions are privaleged, meaning some cannot be trapped, and other techniques must also be used to enable virtualization. *Full virtualization* of unmodified guest kernels was enabled through binary translation, where all sensitive instructions could be translated into privileged instructions. *Paravirtualization* used modifications to the guest operating systems to ensure sensitive operations were trapped. Today, modern hardware architectures include special virtualization instructions which remove the need for binary translation, and additionally remove the need for performance critical shadow structures using two-dimensional hardware page tables [?]. Current virtualization technologies offer a mix of all these techniques, including binary translation for full nested virtualization using Oracle's HVX [?], paravirtualization with Xen [?], and classic trap-and-emulate utilizing modern hardware extensions and the QEMU x86 emulator [?] within the Linux kernel with KVM [?]. Xen is highly used within the research community because of its relatively simple software-only techniques, and KVM is valued for its tight integration with the Linux kernel.

Serverless Computing. Serverless computing has become an increasingly important and desirable platform in the cloud ecosystem. Stemming from the grand vision of computation as a utility, serverless computing offers users the ability to run application code directly on a black-box infrastructure. Serverless has the potential to offer an easy to program, auto-scaling, cost efficient way to utilize cloud infrastructure for users, without the need to manage machine provisioning and configuration or service orchestration [?]. Although there exist many popular industry serverless computing platforms today [?][?][?][?], serverless is still an active area of research, with many improvements to be made [?][?][?].

Cold-start Reduction Efforts. There exists a number





of different techniques which have been proposed in order to reduce cold-start latencies. One broad technique is to reduce the size and complexity of VMs which are used for serverless functions. Because serverless functions running in a cloud infrastructure run on a fairly limited and standard set of physical machines, the number of specialized drivers and kernel modules required to support this hardware is vastly smaller than that in a standard distribution. Additionally, because the function will only need to run a single process, a number of other optimizations can be made to reduce kernel size and boot time. These kinds of operating system optimizations have been explored previously with Unikernels [**?**], the Denali OS [**?**], and are represented in Amazon's custom μ-Kernel and their serverless KVM VMM Firecracker [**?**].

Another technique for reducing VM creation latencies involves the process of *flash-cloning*, in which new VMs are cloned from existing reference VMs. This is an analogous process to forking processes within linux. One work which had success with this technique was the Potemkin Virtual Honeyfarm [**?**]. Lightweight VMs were created as Honeypots for catching security vulnerabilities in cloud applications. This work successfully implements flash-cloning within the Xen hypervisor [**?**] along with copy-on-write memory sharing, but focuses on VM density rather than start-up latency. We implement flash-cloning within a KVM based infrastructure to more closely match with industry standard serverless infrastructure, Amazon Firecracker [**?**], and with the goals of minimizing start-up latency and utilizing native copy-on-write functionality within the Linux kernel.

VM Live Migration. Flash-cloning relates very closely to the more mature virtualization technology of VM live migration [**?**][**?**]. Cloud infrastructure has long required the ability to efficiently migrate VMs across physical hosts. VM live migration in general contains a superset of the functionality needed for flash-cloning, but is focused on efficiently supporting the migration of VM state across a slow network with minimal interruption to the VM. For serverless computing infrastructure, we are focused on extremely low latency and therefore do not want the complex techniques used for live-migration.

5. SUMMARY

In this paper we present *HyperFork*, it's great guys.

6. ACKNOWLEDGMENTS

This is the acknowledgments section

APPENDIX

A. APPENDIX SECTION [RENAME]

This is a section in the appendix