HyperFork: Improving Serverless Latency Through Virtual Machine Flash-Cloning

Michael Colavita David Gardner Mark Wilkening

Harvard University

Abstract

Serverless computing provides a convenient infrastructure for performing computations in response to certain triggers. In most common severless deployments, computations are executed in virtual machines. As a result, the boot time of a virtual machine is a primary component of these services' latency. We propose an alternative paradigm, in which incoming requests trigger a flash clone of an existing, pre-booted virtual machine. Our implementation HyperFork builds on top of KVM and utilizes the copy-on-write functionality of the fork system call to improve performance. We demonstrate that HyperFork improves virtual machine start times by nearly two orders of magnitude while maintaining throughput. We find that copy-on-write performance penalties are minimal for real-world workloads and suggest methods for further improving throughput.

1 Introduction

Serverless Computing. Serverless computing, also known as *Function-as-a-Service* (FaaS), has become an increasingly prevalent platform in the cloud computing ecosystem. In an attempt to realize a vision of computation as a utility, serverless computing allows users to run application code in response to triggers without provisioning infrastructure. In comparison to current cloud computing platforms, clients no longer have to maintain virtual machine images, are billed only for application computation performed in response to requests, and benefit from autoscaling to handle variable request rates. Platforms are available from all major cloud computing companies, including Amazon Lambda, Azure Functions, and Google Cloud Functions. Several significant enterprise customers have moved parts of their services onto serverless platforms, including 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/

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. Prevous work has recorded Amazon Lambda warmstart latencies of around 25ms, and cold-start latencies of 250ms [21]. For cold-start, one significant bottleneck is the creation of a new VM. In the Amazon Firecracker specification [1], new VMs are specified to boot in under 125ms, a significant chunk of the coldstart latency. Pools or buffers of ready VMs may help, but to approach warmstart latencies, improvement in VM creation latencies will be needed. To create a VM, the host Hypervisor/Virtual Machine Monitor (VMM) must initialize virtual resources including CPUs, memory, and other devices, then the guest kernel must be loaded from disk and initialized in guest memory, then the guest kernel must be booted, and finally the function runtime/process can be started and the request handled.

Flash Cloning. Three significant parts of VM creation include (1) loading the kernel into memory, (2) booting the guest kernel, and (3) loading potentially large libraries/runtimes. Management operations within the VMM such as virtual device creation and resource allocation 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.

HyperFork.

To create a new, isolated virtual machine, we propose to re-create only the necessarily distinct VMM components while cloning initialized guest memory and execution context pre-booted virtual machines. From this insight we present *HyperFork*, a KVM-based VM cloning implementation for serverless computing. We demonstrate that HyperFork outperforms standard VM creation latencies by up to two orders of magnitude. We further present a thorough analysis of the potential performance degradation due to copy-on-write memory sharing. We further demonstrate that latency-sensitive workloads display a marked improvement in overall resource utilization without degraded performance.

Our implementation is open source and available on Github at colavitam/hyperfork-kvmtool.

The rest of the paper is organized as follows. Section 2 provides a summary of related work. Section 3 describes the technologies HyperFork is built upon. Section 4 describes the design and architecture of HyperFork. We evaluate its performance in Section 5, discuss the results in Section 6, and then outline current limitations and future directions in Section 7.

2 Related Work

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 [19]. 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 [8]. Current virtualization technologies offer a mix of all these techniques, including binary translation for full nested virtualization using Oracle's HVX [12], paravirtualization with Xen [10], and classic trap-and-emulate utilizing modern hardware extensions and the QEMU x86 emulator [11] within the Linux kernel with KVM [16]. 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 [15]. Although there exist many popular industry serverless computing platforms today [2] [3] [4] [5], serverless is still an active area of research, with many improvements to be made [21] [9] [13].

Cold-start Reduction Efforts. A number of different techniques have been proposed 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. Amazon utilizes such a stripped down kernel in their Firecracker VMM [6]. These kinds of operating system optimizations have also been explored previously in a more extreme form, with Unikernels [18] and the Denali OS [22], where there is no real division between operating system and application code.

Other approaches to efficient resource isolation put the isolation boundary above the operating system level. Containerization technologies such as Docker [**?**] utilize Linux cgroups [**?**] and seccomp mechanism to isolate processes. While these offer very fast startup times compared to Virtual Machines, they are a weaker form of isolation, as processes in containers still share a kernel.

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 [20]. Lightweight VMs were created as Honeypots for catching security vulnerabilities in cloud applications. This work successfully implements flash-cloning within the Xen hypervisor [10] 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 [6], 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 [14] [17]. 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. We rely on many features pioneered by live migration

research in order to efficiently duplicate virtual machine state.

3 Background

In this section we describe the platforms HyperFork is built upon, including an overview of the KVM hypervisor and kvmtool virtual machine monitor.

3.1 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 standard linux process, which contains both VMM management components as well as the running guest. Within the VMM, management components within 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 [6].

3.2 kvmtool

kvmtool [7] is a VMM implementation for KVM with the minimal functionality required to boot a fully functional linux kernel with very 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 [11], 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 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 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 internal VM 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 enables efficient virtualization of the guest and its devices.

4 Implementation

We now describe the architecture and implementation of Hyperfork, along with our design decisions regarding the flash-cloning operation.

For our KVM based virtualization platform, flash-cloning requires duplicating several pieces of VM state. Where possible, we aim to make heavy use of the linux fork operation to duplicate this state, as it is highly optimized and pro-

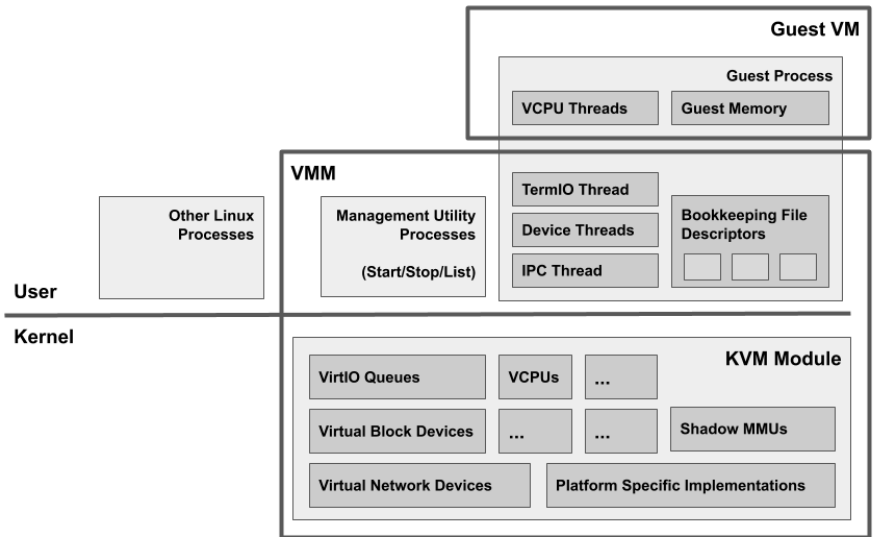

Figure 1: KVM Software Architecture

vides copy-on-write functionality for duplicated pages. Specifically, guest memory will be duplicated with copy-on-write functionality so that the backing pages are not copied for the child unless necessary. However, fork leaves kernelspace management state unduplicated, cannot duplicate all userspace VMM threads, and leaves userspace bookkeeping file descriptors pointing to the parent process's KVM management structures, which are inaccessible in the child. Duplicating kernel state can either be performed within the kernel, by coping in-kernel data structures, or 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.[2] We also add an ad-hoc guest-to-host communication channel with an extension to kvmtool to enable our experimental evaluations.

4.1 Flash-Cloning Support in kvmtool

Our userspace HyperFork implementation for kvmtool proceeds in a series of phases. At a high level, these phases are 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

[2]Note that a kernel implementation would also need to update file descriptors and their memory mappings. This creates quite a mess in practice.

following the fork.

Once the pre-fork routine is complete, kvmtool calls the system call 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[3]
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[4]
7. Attaches the terminal device to a new pseudo-terminal, or detaches it to accept no further input[5]

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.

[3]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.

[4]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.

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

4.2 Guest-to-Host Signaling

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 signaling.

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.

5 Evaluation

We now describe the design of our benchmarks for the kvmtool HyperFork implementation. We have conducted both microbenchmarks, to test the overhead of fork and copy-on-write page duplication, and end-to-end benchmarks to measure throughput and resource utilization improvements.

Experimental Setup. We perform all tests on an m5.metal instance from Amazon Web Services EC2, which features 96 logical processors and 384 GB of memory. The file system image used for all tests is a minimal Linux setup generated using the buildroot utility.[6] This specially generated image contains only the minimal set of utilities and software needed for our benchmarks. In typical use, the guest images used for

[6]buildroot.org

FaaS platforms may be more fully featured and optimized, but such single purpose images have no need for the flexibility of a general purpose Linux distribution image. 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 an idealistic 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.

5.1 Benchmarks

Fork Time. As HyperFork replaces the time taken to boot a virtual machine with the time taken to fork a virtual machine, we run several experiments to test the time for a VM to fork. These tests start a virtual machine, run basic userspace initialization, and then fork. The child VMM marks a timestamp once it finishes restoring KVM state. Figure 5.2 shows the fork time as recorded by the child VM for varying amounts of allocated guest memory.

As memory is duplicated through copy-on-write in a Linux fork, memory is not physically copied to the child process unitl it is modified. However, 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 increase with the amount of memory allocated to the virtual machine.

We compare the results of this fork time benchmark with the time to cold-boot a VM.

Copy-on-Write. Though cloning VMs using fork can decrease VM start times by orders of magnitude, the guest memory mapped in the parent and child virtual machine share physical pages until modified. As a result, modifications to memory in each of the virtual machines can trigger page faults and memory copies that may degrade performance. We implement a benchmark to isolate the performance degradation caused by copy-on-write operations after cloning.

This program begins by allocating some amount of memory, and writes 128 bytes of random data to each 4096-byte page allocated. This forces pages to be allocated by the host kernel. It then signals to the host, which forks the VM. In each child the program then performs an additional pass, again writing random data to each of those pages.

Each of those post-fork writes will trap into the host kernel and induce a copy. To assess copy-on-write overhead, we measure the time for each of these memory passes. We compare passes that trigger a page fault (either due to allocation pre-fork, or copying post-fork). We also compare passes that do not trigger a page fault (both pre-fork and post-fork).

Python Function. In the lifecycle of a typical FaaS request, a VM is started, any relevant state is loaded, and the user-supplied function is run. These services mostly use managed runtimes like NodeJS and Python. To simulate a small latency-sensitive workload, we use the numpy FFT correctness test, which runs in approximately 2.5 seconds. This represents a somewhat long latency sensitive workload, and captures the typical scenario of a function which starts an interpreter, loads external libraries, and performs a CPU-bound computation.

We compare the total time of starting 64 separate VMs, and letting each run the benchmark to completion, with the total time of forking 63 VMs from one reference image and letting them run to completion. We consider the total CPU time savings of starting each of these VMs by forking instead of booting. We also consider a variety of different fork points, including immediately after boot, after python loads, and after packages are loaded. We also verify that benchmark performance is not degraded by the forking process.

5.2 Results

Fork Time. Figure 2 shows our measurement results for our fork and boot time benchmarks. Error bars indicate the 2.5- and 97.5-percentiles.

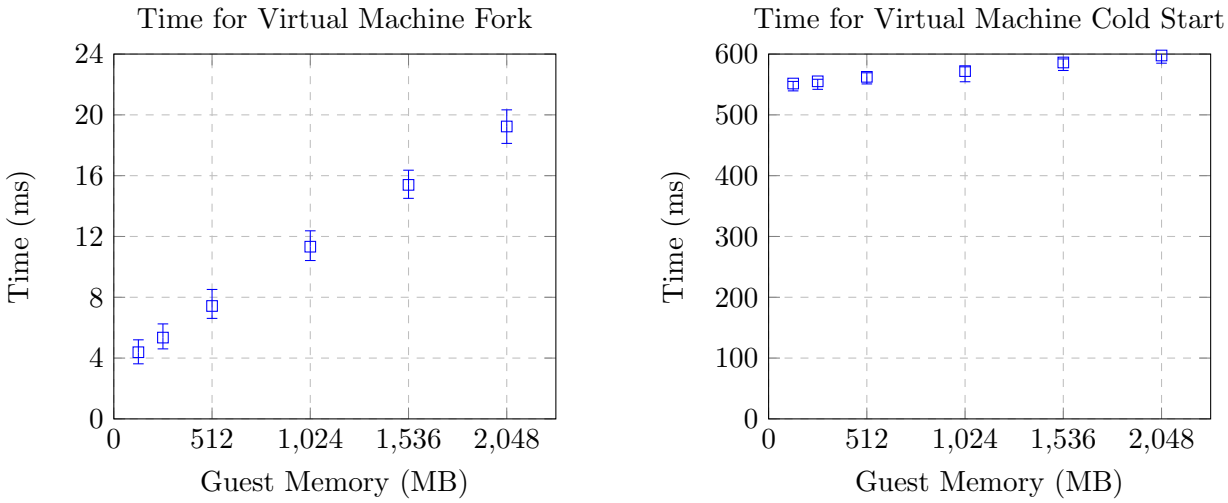

Figure 2: Time for virtual machine cold start vs. fork

Using the small image constructed for our benchmarks, we observed boot times of approximately 563 ms for a 512 MB virtual machine. Boot times increased slightly as memory size increased and overall exhibited very low variance.

The fork operation, on the other hand, took an average of 7.43 ms to complete a fork of a 512 MB virtual machine. This is an improvement of nearly two orders of magnitude. Again, we observed very little variance in our measurements. Fork time appears to scale linearly with the memory size of the virtual machine, likely because the number of page mappings to be duplicated scales linearly with the memory size. However, even at large memory sizes, forking eliminates more than 95% of the boot time.

Copy-on-Write. Figure 3 shows the results of our copy-on-write test. Each graph shows the performance before and after the fork operation. Error bars indicate 2.5- and 97.5-percentiles. When memory writes do not trigger a page fault (left plot), performance does not appear to change before or after a page fault. This means that after pages have been allocated (or reallocated after being copied on write), there is no significant memory performance penalty.

When writes do trigger a page fault (right plot), we observe a considerable difference before and after the fork operation. Before the fork operation, page faults occur when we write to pages that have not yet been allocated. The kernel then allocates a zeroed page and returns. After the fork operation, page faults trigger a page copy. We observe a roughly 28% performance penalty for the memory benchmark when pages must be copied. As expected, the benchmark time scales roughly linearly with the number of pages for which a page fault is triggered.

Python Function. Figure 4 displays the cumulative CPU time required for the python benchmark. The margin of error are negligible (less than 0.2%). We see a significant improvement when forking compared to booting 64 independent virtual machines. Forking after boot immediately results in a 24.8% CPU time savings across the 64 benchmarks. Forking after starting the python interpreter saves 26.7% over not forking. Forking after packages are loaded saves 30.2% over not forking. Thus forking instead of booting new virtual machines can result in significant resource savings for servers running short duration or latency sensitive serverless workloads. This effect would be accentuated for shorter jobs and more minor for longer ones.

Furthermore, figure 5 shows the mean benchmark completion times for each forking scheme. Interestingly, we do not observe any performance degradation when forking instead of booting. In fact, forking exhibits a slight (but statistically significant) reduction in benchmark completion

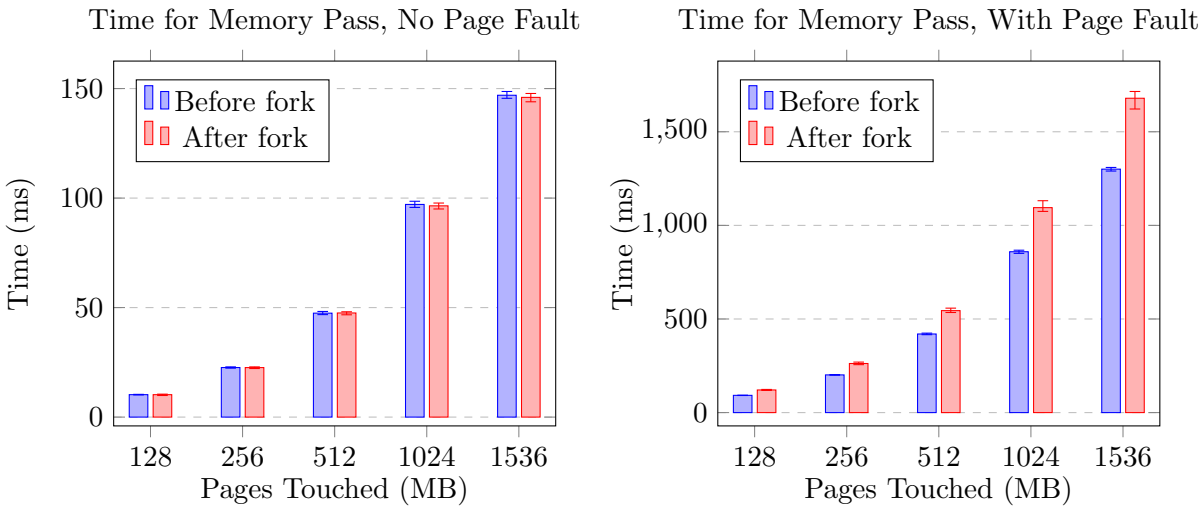

Figure 3: Memory benchmark with and without copy-on-write ($n = 100$)

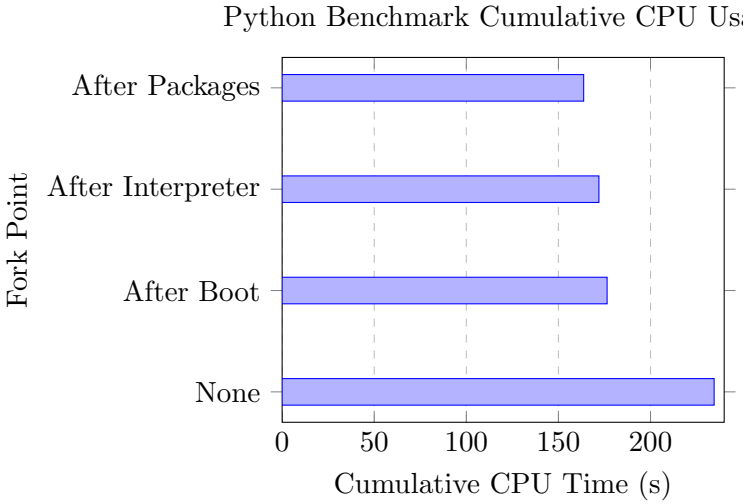

Figure 4: Cumulative CPU time for 64 runs of python benchmark

Fork Point	Benchmark Time (ms)	Relative Time
None	2500 ± 9.14	100%
After Boot	2433 ± 8.36	97.3%
After Interpreter	2409 ± 8.36	96.4%
After Packages	2401 ± 8.04	96.0%

Figure 5: Mean benchmark completion time (with 95% confidence interval)

time. We suspect this may be due to increased memory locality due to shared pages after forking.

6 Discussion

7 Future Work

Current Limitations. While our current HyperFork implementation provides a reliable fork primitive for our basic benchmarks, it has several limitations that would need to be addressed before production deployment. Currently, only read-only root filesystems are supported. Read-write filesystems can cause corruption when multiple forked virtual machines have contradictory state. We observed this several times in practice. This could be addressed by adding an overlay layer to the block device driver that stores writes to the file system in VMM memory instead of persisting them to disk, making writes from different virtual machines invisible to each other.

We have not added support for all of kvmtool's devices to Hyperfork. Specifically, the network device and virtio balloon device are not supported. Additionally, kvmtool has an outstanding bug in its terminal emulation that manifests in degraded performance after forking. We have worked around this in our current implementation.

Additionally, there is an outstanding race condition in the kvm-clock feature, a paravirtualized clock accessible from the guest virtual machine. This manifests itself as occasional non-monotonicity in the guest, causing prolonged kernel-level stalls in the guest virtual machine. For our tests, we have disabled kvm-clock to avoid these problems. We suspect it can be resolved by careful adjustment of the clock when restoring virtual machine state.

Additional Research.

- HyperFork currently does not handle very many devices. A deployable HyperFork would need to re-initialize and re-configure network state post-fork. We also assume in our implementation that the guest is running entirely from a RAM filesystem. If any external virtual or physical disks were in use, then HyperFork would need to address synchronization concerns with those devices.

- Currently, HyperFork operates entirely in userspace by serializing all KVM state pre-fork and then recreating it post-fork. We suspect that a kernel-mode implementation may offer further performance benefits. This avoids the overhead of copying KVM state into userpace and then back into the kernel, instead just passing the state directly between the KVM backing structures for the parent and child processes.

- There are several serious security concerns with cloning Virtual Machines in production. ASLR and KASLR are defeated, since the guest memory is copied exactly. It is also not desirable for two guest VMs to share a source of randomness, so any random generators would need to be re-seeded in the child VM.

- Firecracker boasts significant boot time improvements over existing hypervisor solutions. In our experiments we discovered that the configuration of the guest kernel can also affect boot times by an order of magnitude. We therefore would like to investigate whether Firecracker's performance improvements come from its inherent design and implementation or if it gains these benefits primarily by using stripped-down guest kernels and filesystem images.

8 Conclusion

References

[1] `https://github.com/firecracker-microvm/firecracker/blob/master/SPECIFICATION.md`, accessed 2019.

[2] https://aws.amazon.com/lambda/, accessed 2019.

[3] https://cloud.google.com/functions/, accessed 2019.

[4] https://azure.microsoft.com/en-us/services/functions/, accessed 2019.

[5] https://openwhisk.apache.org/, accessed 2019.

[6] https://lwn.net/Articles/775736/, accessed 2019.

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

[8] K. Adams and O. 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] I. Baldini, P. Cheng, S. J. Fink, N. Mitchell, V. Muthusamy, R. Rabbah, P. Suter, and O. Tardieu. The serverless trilemma: Function composition for serverless computing. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2017, pages 89–103, New York, NY, USA, 2017. ACM.

[10] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. 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.

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

[12] A. Fishman, M. Rapoport, E. Budilovsky, and I. Eidus. HVX: Virtualizing the cloud. In *Presented as part of the 5th USENIX Workshop on Hot Topics in Cloud Computing*, San Jose, CA, 2013. USENIX.

[13] J. M. Hellerstein, J. M. Faleiro, J. E. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, and C. Wu. Serverless computing: One step forward, two steps back. *CoRR*, abs/1812.03651, 2018.

[14] M. R. Hines and K. 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.

[15] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar, J. Gonzalez, R. Ada Popa, I. Stoica, and D. A. Patterson. Cloud programming simplified: A berkeley view on serverless computing, 02 2019.

[16] A. Kivity Qumranet, Y. Kamay Qumranet, D. Laor Qumranet, U. Lublin Qumranet, and A. Liguori. Kvm: The linux virtual machine monitor. *Proceedings Linux Symposium*, 15, 01 2007.

[17] H. A. Lagar-Cavilla, J. A. Whitney, A. M. Scannell, P. Patchin, S. M. Rumble, E. de Lara, M. Brudno, and M. 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.

[18] A. Madhavapeddy and D. J. Scott. Unikernels: Rise of the virtual library operating system. *Queue*, 11(11):30:30–30:44, Dec. 2013.

[19] G. J. Popek and R. P. Goldberg. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17(7):412–421, July 1974.

[20] M. Vrable, J. Ma, J. Chen, D. Moore, E. Vandekieft, A. C. Snoeren, G. M. Voelker, and S. 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.

[21] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift. Peeking behind the curtains of serverless platforms. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '18, pages 133–145, Berkeley, CA, USA, 2018. USENIX Association.

[22] A. Whitaker, M. Shaw, and S. 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.