Title of the Paper

First Last First Last First Last

Note: Make sure all info is hidden for double-blind submissions 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: Coldstart. 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 [19]. 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.

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.

2. HYPERFORK ARCHITECTURE

2.1 KVM Overview

TODO: KVM description

2.2 `kvmtool`

`kvmtool` is a userland VMM with the minimal functionality required to boot a functional linux kernel with very basic virtualization devices. These devices include `virtio` block, network, filesystem, balloon, hardware random number generation, and console devices, along with a legacy 8250 serial device. (CITE: kvmtool) It 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 userland VMM as it offers a very similar set of functionality to Firecracker, the VMM used to power Amazon Lambda. (TODO: contrast firecracker and kvmtool) However, 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 a serverless platform.

To virtualize efficiently, `kvmtool` makes use of a large number of threads for managing vCPUs and emulated devices. When the virtual machine is started, `kvmtool` creates a thread for each class of device, including the terminal, 8250 serial console, block devices, and `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 large set of threads enable efficient virtualization of the guest and its devices.

2.2.1 *HyperFork in* `kvmtool`

Our userland HyperFork implementation for `kvmtool` proceeds in a series of phases. 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 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 performs the following:

1. Saves individual vCPU state, for each vCPU (registers, interrupt configuration, etc.)
2. Saves global vCPU state (interrupt configuration, clock)
3. 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:

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]

One the post-fork routine is complete, the vCPUs begin executing in the child and the fork is complete.

2.3 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

[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.

completed. As the guest's userland 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 userland through the `outb` functions in the C standard library (CITE: outb, linux man). 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. For benchmarking, we include a `fork` and `done` executable that signal the two events. 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. The experimental setup includes information about the following 5 things: (i) *Experimental Platform*, i.e., the machine we used, (ii) *Implementation*, i.e., whether it is a prototype system, based on an existing one, and any other relevant detail, (iii) *Configuration*, i.e., the system-specific configuration and tuning that might have been needed, and discuss the possible values for any parameter. (iv) *Workloads*, i.e., dataset characteristics, and query characteristics, and (unless the whole paper is a single experiment this here just gives a summary of the workloads used and each experiment later on gives the exact setup) (v) *Metrics*, typically latency and/or throughput with some details and the methodology.

For the experimental platform we do not want to use exactly the same phrasing in every paper, however, the content should be the same.

We perform all tests on [SOME MACHINE], with [SOME SPECS].

3.1 Benchmarks

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 many pages of memory, and writing a small amount 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.

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. 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.

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 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 traped by the hardware, and emulated safely within the VMM using shadow structures for privaleged state [17]. In x86 however, not all sensitive instructions are privaleged, meaning they 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 privaleged 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 [6]. Current virtualization technologies offer a mix of all these techniques, including binary translation for full nested virtualization using Oracle's HVX [10], paravirtualization with Xen [8], and classic trap-and-emulate utilizing modern hardware extensions and the QEMU x86 emulator [9] within the Linux kernel with KVM [14]. 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 desireable 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 [13]. Although there exist many popular industry serverless computing platforms today [1][2][3][4], serverless is still an active area of research, with many improvements to be made [19][7][11].

Flash Cloning / MicroVMs / Coldstart reduction efforts. Include Potemkin and Snowflock, as well as that short one about caching python modules. Also maybe include unikernels/Denali/Firecracker. [18][15][16][20][5]

VM Live Migration. Cite some state-of-the-art research / survey, maybe some tools/implementations. Focus text on explaining difference from flash cloning. [12]

5. SUMMARY

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

6. ACKNOWLEDGMENTS

This is the acknowledgments section Note: Decide whether to add ack's in the submission

Note: Make sure we have the updated version of the conference style

7. REFERENCES

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

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

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

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

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

[6] 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.

[7] 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.

[8] 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.

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

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

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

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

[18] 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.

[19] 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.

[20] 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.

APPENDIX

A. APPENDIX SECTION [RENAME]

This is a section in the appendix

B. LIST OF FIXMES

List of Corrections