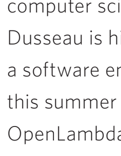
SOCK: Serverless-Optimized Containers

EDWARD OAKES, LEON YANG, DENNIS ZHOU, KEVIN HOUCK, TYLER CARAZA-HARTER, ANDREA C. ARPACI-DUSSEAU, AND REMZI H. ARPACI-DUSSEAU

Edward Oakes holds a BS from the University of Wisconsin-Madison in computer science and is an incoming PhD student at the University of California-Berkeley. As an undergraduate, he was advised by professors Andrea and Remzi Arpaci-Dusseau and is a primary contributor to the OpenLambda project. oakes@cs.wisc.edu

Leon Yang received his bachelor's degree from the University of Wisconsin-Madison, where he is currently pursuing a master's degree in computer science. Professor Remzi Arpaci-Dusseau is his adviser. He will be working as a software engineering intern at Facebook this summer and is a contributor to the OpenLambda project. gyang48@wisc.edu


Dennis Zhou is a recent MS graduate from the University of Wisconsin-Madison. He was advised by Andrea and Remzi Arpaci-Dusseau and worked on OpenLambda at the Microsoft Gray Systems Lab. This summer, he is joining Facebook as a Software Engineer working on Linux. dennisszhou@gmail.com

Kevin Houck is a recent Bachelor of Science graduate in computer science at the University of Wisconsin-Madison. He was advised by Professor Aditya Akella and has previously contributed to OpenLambda. This summer he will be continuing ongoing research in serverless computing and this fall will join Amazon as a Software Engineer. houck@cs.wisc.edu

Serverless computing is becoming increasingly popular as a way to avoid paying for idle periods and gracefully handle load spikes. Serverless platforms typically use containers to isolate lambda instances. General-purpose container systems such as Docker, however, are not well suited to serverless sandboxing and introduce unnecessary startup costs. In this work, we analyze the tradeoffs offered by alternative containerization primitives and use our findings to build a lean container system, SOCK, optimized for serverless workloads. Replacing Docker with SOCK in the OpenLambda serverless platform results in an 18x speedup.

The effort to maximize developer velocity has greatly influenced the way programmers write and run their code. Developers are writing code in higher-level languages, such as JavaScript and Python, and reusing libraries when possible in order to avoid memory management details and the re-implementation of common logic. Developers are also decomposing their applications into cooperating microservices, easing maintenance burdens and making incremental development simpler.

Containers are an increasingly popular way to deploy these microservices. Instead of virtualizing low-level resources (e.g., network interfaces), containers virtualize high-level resources (e.g., port numbers). Containers thus serve as a lightweight alternative to virtual machines, providing each microservice with a virtualized environment and eliminating the need to provision a different operating system for each microservice.

Recently, *serverless computation* has emerged as a new style of cloud platform that integrates a common development approach (application decomposition) with a popular deployment strategy (auto-scaling containers). In various serverless offerings, such as AWS Lambda [3], developers decompose their applications into handlers, called *lambdas*, that execute in response to web requests or other events. Lambda instances execute inside sandboxes (typically containers) and automatically scale up or down based on load. Leaving both the runtime and autoscaling to the platform, developers no longer need to manage servers themselves, hence the name "serverless." New instances are provisioned quickly (often in less than a second), and tenants are only billed during the handling of events, making serverless ideal for load bursts as well as cost savings during application idleness.

The Problem. While high-level languages, reusable libraries, containers, and serverless platforms all improve developer velocity, these approaches also create new infrastructure problems by making process cold-start more frequent and expensive. Languages like Python and JavaScript require heavy runtimes, making startup over 10x slower than launching an equivalent C program [1]. Reusing code introduces further startup latency from library loading and initialization [4]. Running microservices in separate containers, rather than just separate processes, introduces a variety of additional initialization overheads [7]. Serverless computing multiplies these costs: if a monolithic application is decomposed to *N* lambda handlers, startup frequency is similarly amplified.

Tyler Caraza-Harter completed his PhD at the University of Wisconsin-Madison in 2016, where he was advised by professors Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau and did research on containers and serverless computing. After graduation, he worked on Azure SQL at Microsoft Gray Systems Lab, and is returning this fall to UW-Madison as an instructor. Tyler is actively involved in two open-source projects, the Pivot Libre project for preferential voting (https://github.com/pivot-libre) and the OpenLambda project (https://github.com/open-lambda). tylerharter@gmail.com

Andrea Arpaci-Dusseau is a Full Professor of Computer Sciences at the University of Wisconsin-Madison. She is an expert in file and storage systems, having published more than 80 papers in this area, co-advised 24 PhD students, and received 11 best paper awards; for her research contributions, she was recently recognized with a UW-Madison Vilas Mid-Career Investigator award. She also created a service-learning course in which UW-Madison students teach CS to more than 200 elementary-school children each semester. dusseau@cs.wisc.edu

Remzi Arpaci-Dusseau is a Full Professor in the Computer Sciences Department at the University of Wisconsin-Madison. He co-leads a group with his wife, Professor Andrea Arpaci-Dusseau. They have graduated 24 PhD students in their time at Wisconsin, won 11 best-paper awards, and some of their innovations now ship in commercial systems and are used daily by millions of people. Remzi has won the SACM Student Choice Professor of the Year award four times, the Carolyn Rosner "Excellent Educator" award, and the UW-Madison Chancellor's Distinguished Teaching Award. Chapters from a freely available OS book he and Andrea co-wrote, found at http://www.ostep.org, have been downloaded millions of times in the past few years. remzi@cs.wisc.edu

Why, exactly, is it so slow to start containerized Python programs that have dependencies?

In order to answer that question, we embark on two performance analysis studies. First, we take a look at Linux containers, which are typically based on Linux namespaces and other abstractions. By instrumenting the kernel and isolating specific aspects of containerization (e.g., container storage), we identify several bottlenecks. For example, network namespaces are not scalable due to a single large lock in the kernel, leading to long latencies when many containers are created concurrently. Second, we study how Python programs use libraries in an analysis of 876K Python projects scraped from GitHub and 101K unique packages downloaded from the popular PyPI repository. We find that many popular packages take 100 ms to import, and installing them can take seconds.

We leverage the findings from these two studies to build a new special-purpose container system, SOCK (roughly for serverless optimized containers), that streamlines cold-start initialization for Python code that has library dependencies. We integrate SOCK with the OpenLambda serverless platform [5] to support modern development patterns, without incurring excessive startup latencies. SOCK uses lightweight isolation primitives, avoiding the performance bottlenecks identified in our Linux primitive study, to achieve an 18x speedup over the general-purpose Docker container system. SOCK also provisions new containers using a new approach that generalizes zygote initialization, a strategy introduced by Android for Java processes.

In an image-resizing case study, these strategies help SOCK reduce cold-start platform overheads by 2.8x and 5.3x relative to the AWS Lambda and OpenWhisk serverless platforms, respectively.

More results from our two performance studies and details about SOCK can be found in [9].

Breaking Down Container Performance

Namespaces are the key abstraction in Linux for logically isolating resources. Namespaces virtualize resources by allowing different containers to use the same virtual name, mapped to distinct physical resources on the host. For example, *network namespaces* allow different containers to use the same virtual port number (e.g., 80), backed by different physical ports on the host (e.g., 8080 and 8081). Similarly, *mount namespaces* give containers access to their own virtual file system roots, backed by different physical directories in the host. Linux also provides namespaces for UTS, IPC, PID, and other resources.

An `unshare` system call allows a process to create and switch to a new set of namespaces. Arguments to `unshare` allow careful selection of which resources need new namespaces. Namespaces are automatically reaped when the last process using them exits.

The flexibility of `unshare` allows us to study the performance and scalability of the various namespaces, used independently or in conjunction. Combining the performance numbers with measurements from kernel instrumentation revealed two scalability bottlenecks, in the network and mount namespaces.

During creation of a network namespace, Linux iterates over all existing namespaces while holding a global lock, searching for namespaces that should be notified of the configuration change. Thus, costs increase proportionally as more namespaces are created. Network namespaces are the primary bottleneck preventing high throughput of concurrent calls to `unshare`.

Figure 1 shows the impact of network namespaces on overall creation/deletion throughput (i.e., with five namespaces). With unmodified network namespaces, throughput peaks at about 200 c/s (containers/second). With minor optimizations (disabling IPv6 and eliminating

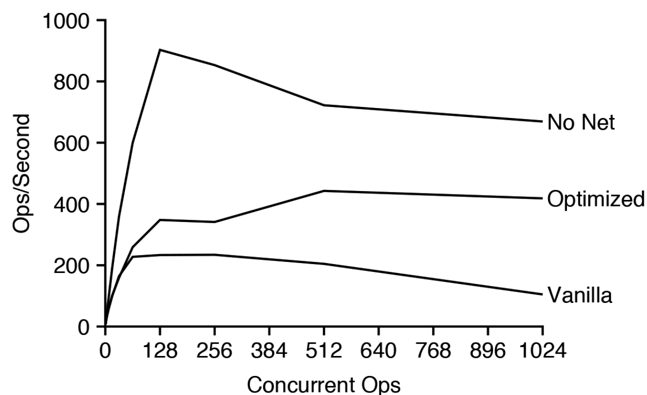


Figure 1: Network namespaces

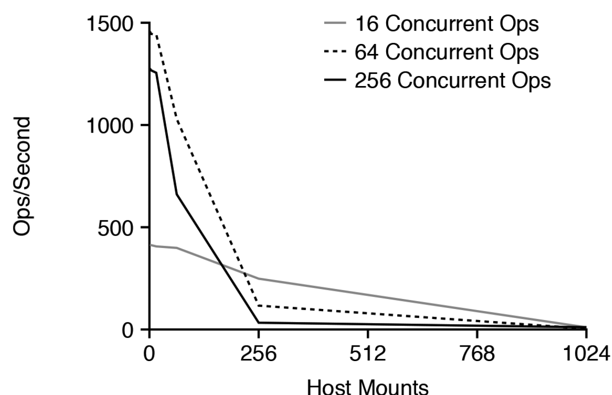


Figure 2: Mount namespaces

the costly broadcast code), it is possible to churn over 400 c/s. However, eliminating network namespaces entirely provides throughput of 900 c/s.

In contrast to network namespaces, it is possible to concurrently create many mount namespaces. However, mount namespaces scale poorly with the number of preexisting mount points on the host, as each new mount namespace starts as a copy of the host's mount points. Figure 2 illustrates this problem: if there are few mount points on the host, we can create nearly 1500 mount namespaces per second. However, as the number of host mounts grows large, the rate at which namespaces can be cloned approaches zero.

Implications. The `unshare` system call provides significant flexibility over which namespaces are used for containers. Depending on the use case, not every namespace may be necessary, so costly namespaces (e.g., those for the network and mount points) should be avoided when possible. Network namespaces are useful for servers that listen on a port, but are less applicable for serverless lambdas that take input from the framework and typically run behind a Network Address Translation layer. Mount namespaces provide a flexible mechanism for exposing specific host mount points inside a container, but in simpler scenarios, the older `chroot` Linux system call may be a better option for isolating the file system. Using `chroot` is essentially free, with calls taking less than one microsecond.

The Cost of Reusing Code

Even if lambdas are executed in lightweight sandboxes, reusing code by relying on various packages can make cold start slow, because the libraries must be re-imported and initialized every time lambda instances are rebalanced or scale up [10].

In order to understand these library-related costs, we scrape and analyze 876K Python projects from GitHub. We expect that few of these applications currently run as lambdas; however, our goal is to identify potential obstacles that may prevent them from being ported to lambdas in the future. We extract likely dependencies in the projects on packages in the popular Python Package Index (PyPI) repository, resolving naming ambiguity in favor of more popular packages. We find that 36% of imports are to just 20 popular packages, shown along the x-axis in Figure 3.

If one of these package is being used for the first time (by a lambda instance or in some other scenario), it will be necessary to *download* the package over the network (possibly from a nearby mirror), *install* it to local storage, and *import* the library to Python bytecode. Some of these steps may be skipped upon subsequent execution, depending on the platform. Figure 3 shows these costs for each of the packages. Fully initializing a package takes 1 to 13 seconds. Every part of the initialization is expensive on average: downloading takes 1.6 seconds, installing takes 2.3 seconds, and importing takes 107 ms.

Implications. Many modern applications, such as Gmail, regularly experience request latency in the tens of milliseconds (including Internet RTT) [5]. If such applications are ported to serverless platforms, even the smallest library-initialization cost (i.e., importing) will dominate, to say nothing of download and install costs that could be necessary. Circumventing these overheads will be key to making serverless a viable option to such latency-sensitive applications.

Serverless Containers

We now describe our design and implementation of SOCK, a container system optimized for use in serverless platforms. SOCK carefully avoids the bottlenecks identified in our analysis of container performance. We integrate SOCK with the OpenLambda serverless platform, replacing general-purpose Docker containers as the primary sandboxing mechanism for OpenLambda. We use additional pools of SOCK containers to construct a caching system that helps lambda instances avoid the startup latencies identified in our study of Python libraries.

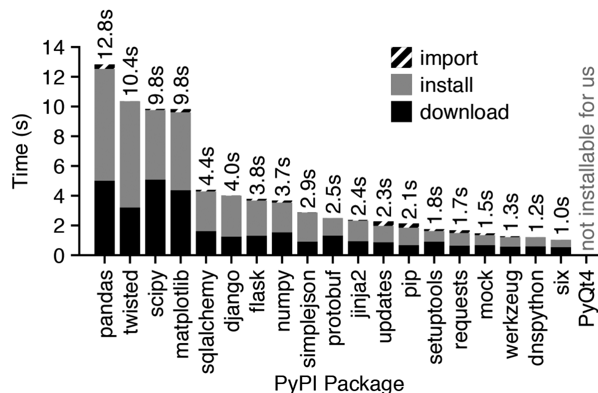


Figure 3: Package initialization costs

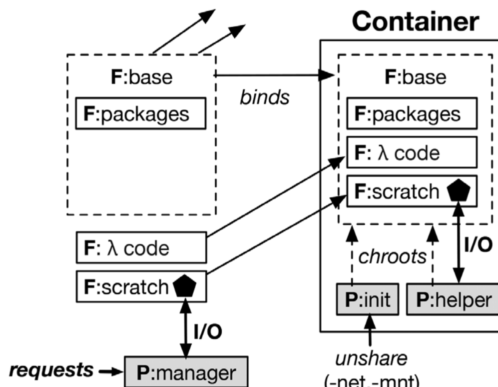


Figure 4: Lean containers

Lean Containers

Figure 4 shows how SOCK efficiently and securely creates containers without requiring costly mount or network namespaces. An init process ("P:init") calls `unshare` to create the necessary namespaces. A second helper process ("P:helper") joins the namespaces later and is responsible for forwarding events and requests to the lambda handler.

Provisioning container storage involves first populating a directory on the host to use as a container root. SOCK stitches together a root directory using several bind mounts. A bind mount efficiently makes a directory at one location in the host file system appear at a second location. SOCK first bind mounts a base directory ("F:base") containing an Ubuntu installation as read-only to serve as a container root; we can afford to back this by a RAM disk as every handler is required to use the same base. A directory used for package caching ("F:packages") is then mounted over the base, as described later. The same base and packages are read-only shared in every container. SOCK finally binds handler code ("F: λ code") as read-only and a scratch directory ("F:scratch") as writable in every container.

The initial processes running in the container (i.e., "P:init" and "P:helper" in Figure 4) call `chroot` to use the populated directory as the container's root file system. We do not require other host mounts in the container, so SOCK avoids the costly creation of a new mount namespace for the container.

The scratch-space mount of every SOCK container contains a UNIX domain socket (the black pentagon in Figure 4) that is used for communication between the OpenLambda manager and processes inside the container. Event and request payloads received by OpenLambda are forwarded over this channel. Thus, lambda instances do not need to listen for input on network ports, so we avoid using poor-scaling network namespaces.

Generalized Zygotes

Zygote provisioning is a technique where new processes are started as forks of an initial process, the zygote, that has already pre-imported various libraries likely to be needed by applications. Linux's copy-on-write sharing reduces the memory consumption of the forked child processes and saves them from all needing to perform the same library initialization work. Zygotes were first introduced on Android systems for Java applications [4].

We implement a more general zygote-provisioning strategy for SOCK. Specifically, SOCK zygotes differ as follows: (1) the set of pre-imported packages is determined at runtime based on usage; (2) SOCK scales to very large package sets by maintaining multiple zygotes with different pre-imported packages; (3) provisioning is fully integrated with containers; and (4) processes are not vulnerable to malicious packages they did not import.

The key challenge to using zygotes for SOCK is integration with containers. We do not trust either lambda handler code, or the package code that handlers may import, so both zygote processes and handlers are containerized. Landing a forked child process in a new container, distinct from the container housing the zygote process, requires a non-trivial relocation protocol described in detail in [9].

Serverless Caching

We use SOCK to build a three-tier caching system for OpenLambda, shown in Figure 5. First, a *handler cache* maintains idle handler containers in a paused state; the same approach is taken by AWS Lambda [3]. Paused containers cannot consume CPU, and unpausing is faster than creating a new container; however, paused containers consume memory, so SOCK limits total consumption by evicting paused containers from the handler cache on an LRU basis.

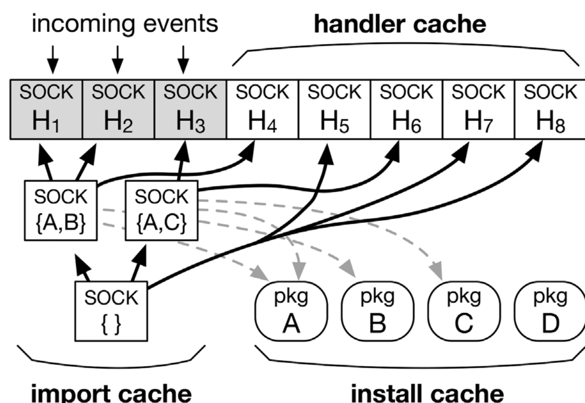


Figure 5: Serverless caching

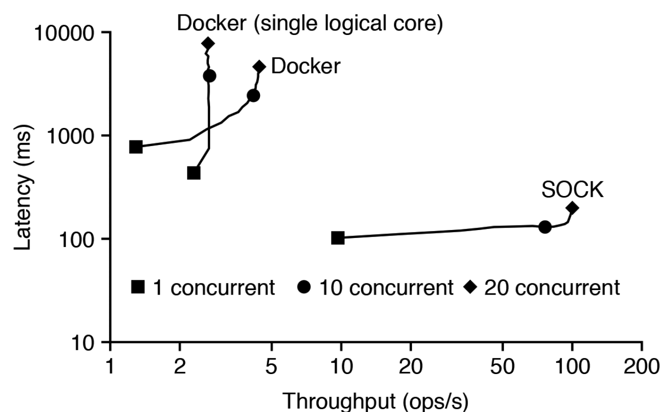


Figure 6: Docker vs. SOCK. Request throughput (x-axis) and latency (y-axis) are shown for SOCK (without zygotes) and Docker for varying concurrency.

Second, an *install cache* contains a large, static set of pre-installed packages on disk. This installation is mapped read-only into every container for safety. Some of the packages may be malicious, but they do no harm unless a handler chooses to import them.

Third, an *import cache* is used to manage zygotes. We have already described a general mechanism for creating many zygote containers, with varying sets of packages pre-imported. However, zygotes consume memory, and package popularity may shift over time, so SOCK decides the set of zygotes available based on the import-cache policy.

In addition to deciding when to add or remove entries from the cache, the import-cache policy needs to decide which zygote to use as the parent from which to fork a child process to serve as the lambda instance. In this regard, the SOCK cache is fundamentally different from traditional caches. Lookup in a traditional cache returns in a hit or miss. SOCK caches never miss and always return one or more hits. Even in the worst case, SOCK can provision a new process by forking a simple Python interpreter with no libraries pre-imported. Or, in the more useful case, there may be multiple zygotes, with varying subsets of the necessary packages pre-imported.

In general, SOCK attempts to choose a zygote that pre-imports a larger subset of the required libraries. This minimizes the number of libraries that a child must import after it is forked from the parent zygote.

One tempting policy to improve performance when possible is to choose zygotes that import a superset of the packages needed by a handler. The child process would then need to import nothing after it is forked. However, we assume the packages may be malicious; pre-importing a library that a handler does not want would expose the handler to a new threat. Thus, for safety, SOCK only chooses zygotes that have imported subsets of the required packages.

Performance Comparisons

We now evaluate the performance of SOCK's lean containers relative to Docker-based OpenLambda and other platforms.

SOCK avoids many of the expensive operations, such as network namespaces, necessary to construct a general-purpose container. In order to evaluate the benefit of lean containerization, we concurrently invoke no-op lambdas on OpenLambda, using either Docker or SOCK as the container engine. We disable all SOCK caches and zygote preinitialization. We run this experiment on two machines, a package mirror and an OpenLambda worker. The machines have 8-core 2.0 GHz Xeon D-1548 processors and 64 GB of RAM. We allocate 5 GB of memory for the handler cache and 25 GB for the import cache.

Figure 6 shows the request throughput and average latency as we vary the number of concurrent outstanding requests. SOCK is strictly faster on both metrics, regardless of concurrency. For 10 concurrent requests, SOCK has a throughput of 76 requests/second (18x faster than Docker) with an average latency of 130 milliseconds (19x faster).

Some of the namespaces used by Docker rely heavily on RCU synchronization, which provides a read-optimized locking mechanism. RCU usage scales poorly with the number of cores [8]. Figure 6 also shows Docker performance with only one logical core enabled: relative to using all cores, this reduces latency by 44% for concurrency = 1, but throughput no longer scales with concurrency.

In addition to streamlining the container-creation protocol, SOCK provisions Python interpreters from zygotes with pre-imported packages. We evaluate these mechanisms with a real-world case study: on-demand image resizing [6]. We write a lambda that reads an image from AWS S3, uses the Pillow package to resize it, and writes the output back to S3. For this experiment, we compare SOCK to AWS Lambda [3] and OpenWhisk [2],

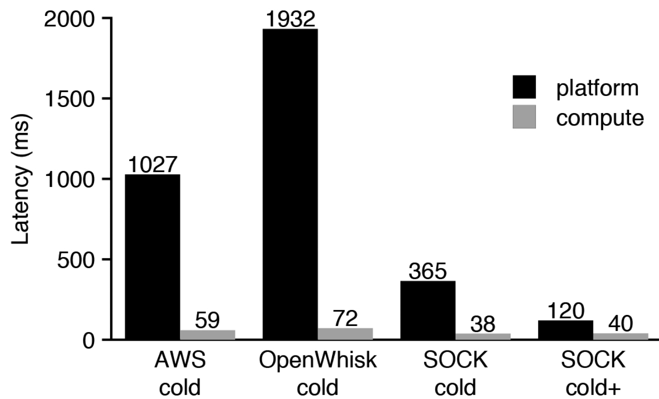


Figure 7: AWS Lambda and OpenWhisk. Platform and compute costs are shown for cold requests to an image-resizing lambda. S3 latencies are excluded to minimize noise.

using 1 GB lambdas (for AWS Lambda) and a pair of m4.xlarge AWS EC2 instances (for SOCK and OpenWhisk); one instance services requests and the other hosts handler code.

For SOCK, we preinstall Pillow and the AWS SDK (for S3 access) to the install cache and specify these as handler dependencies. For AWS Lambda and OpenWhisk, we bundle these dependencies with the handler itself, inflating the handler size from 4 KB to 8.3 MB. For each platform, we exercise cold-start performance by measuring request latency after re-uploading our code as a new handler. We instrument handler code to separate compute and S3 latencies from platform latency.

The first three bars of Figure 7 show compute and platform results for each platform. "SOCK cold" has a platform latency of 365 ms, 2.8x faster than AWS Lambda and 5.3x faster than OpenWhisk. "SOCK cold" compute time is also shorter than the other compute times because all package initialization happens after the handler starts running for the other platforms, but SOCK performs package initialization work as part of the platform. The "SOCK cold+" represents a scenario similar to "SOCK cold," where the handler is being run for the first time but a different handler that also uses the Pillow package has recently run. This scenario further reduces SOCK platform latency by 3x to 120 ms.

Conclusion

Serverless platforms promise cost savings and extreme elasticity to developers. Unfortunately, these platforms also make initialization slower and more frequent, so many applications and microservices may experience slowdowns if ported to the lambda model. In this work, we identified container initialization and package dependencies as common causes of slow lambda startup. Based on our analysis, we built SOCK, a streamlined container system optimized for serverless workloads that avoids major kernel bottlenecks. We further generalized zygote provisioning and built a package-aware caching system. Our hope is that this work, alongside other efforts to minimize startup costs, will make serverless deployment viable for an ever-growing class of applications.

Acknowledgments

Feedback from anonymous reviewers has significantly improved this work. We also thank the members of ADSL and our colleagues at GSL for their valuable input.

This material was supported by funding from NSF grant CNS-1421033, DOE grant DESC0014935, and student funding from Microsoft. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and may not reflect the views of NSF, DOE, Microsoft, or other institutions.

References

[1] I. E. Akkus, R. Chen, I. Rimac, M. Stein, K. Satzke, A. Beck, P. Aditya, and V. Hilt, "SAND: Towards High-Performance Serverless Computing," in 2018 USENIX Annual Technical Conference (USENIX ATC '18).

[2] Apache OpenWhisk: https://openwhisk.apache.org/.

[3] AWS Lambda: https://aws.amazon.com/lambda/.

[4] D. Bornstein, "Dalvik Virtual Machine Internals," talk at Google I/O, 2008: https://www.youtube.com/watch?v=ptjedOZEXPM.

[5] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Serverless Computation with OpenLambda," in *Proceedings of the 8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud '16)*: https://www.usenix.org/system/files/conference/hotcloud16/hotcloud16_hendrickson.pdf.

[6] B. Liston, "Resize Images on the Fly with Amazon S3, AWS Lambda, and Amazon API Gateway," AWS Compute Blog: https://aws.amazon.com/blogs/compute/resize-images-on-the-fly-with-amazon-s3-aws-lambda-and-amazon-api-gateway, January 2017.

[7] F. Manco, C. Lupu, F. Schmidt, J. Mendes, S. Kuenzer, S. Sati, K. Yasukata, C. Raiciu, and F. Huici, "My VM Is Lighter (and Safer) than Your Container," in *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*, pp. 218–233: http://cnp.neclab.eu/projects/lightvm/lightvm.pdf.

[8] P. E. McKenney, "Introduction to RCU Concepts: Liberal Application of Procrastination for Accommodation of the Laws of Physics for More Than Two Decades!" LinuxCon Europe 2013.

[9] E. Oakes, L. Yang, D. Zhou, K. Houck, T. Harter, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, "SOCK: Rapid Task Provisioning with Serverless-Optimized Containers," 2018 USENIX Annual Technical Conference (USENIX ATC '18).

[10] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, "Peeking Behind the Curtains of Serverless Platforms," 2018 USENIX Annual Technical Conference (USENIX ATC '18).

USENIX Supporters

USENIX Patrons
Facebook • Google • Microsoft • NetApp • Private Internet Access

USENIX Benefactors
Amazon • Bloomberg • Oracle • Squarespace • VMware

USENIX Partners
Booking.com • CanStockPhoto • Cisco Meraki
DealsLands • Fotosearch • thebestvpn.com

Open Access Publishing Partner
PeerJ