SOCK: Rapid Task Provisioning with Serverless-Optimized Containers

Edward Oakes, Leon Yang, Dennis Zhou, and Kevin Houck, *University of Wisconsin-Madison;* Tyler Harter, *Microsoft, GSL;* Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau, *University of Wisconsin-Madison*

https://www.usenix.org/conference/atc18/presentation/oakes

This paper is included in the Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC '18).

July 11–13, 2018 • Boston, MA, USA

ISBN 978-1-931971-44-7

SOCK: Rapid Task Provisioning with Serverless-Optimized Containers

Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter[†],
Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau

University of Wisconsin–Madison [†] Microsoft Gray Systems Lab

Abstract

Serverless computing promises to provide applications with cost savings and extreme elasticity. Unfortunately, slow application and container initialization can hurt common-case latency on serverless platforms. In this work, we analyze Linux container primitives, identifying scalability bottlenecks related to storage and network isolation. We also analyze Python applications from GitHub and show that importing many popular libraries adds about 100 ms to startup. Based on these findings, we implement SOCK, a container system optimized for serverless workloads. Careful avoidance of kernel scalability bottlenecks gives SOCK an 18× speedup over Docker. A generalized-Zygote provisioning strategy yields an additional 3× speedup. A more sophisticated three-tier caching strategy based on Zygotes provides a 45× speedup over SOCK without Zygotes. Relative to AWS Lambda and OpenWhisk, OpenLambda with SOCK reduces platform overheads by 2.8× and 5.3× respectively in an image processing case study.

1. Introduction

The effort to maximize developer velocity has driven many changes in the way programmers write and run their code [43]. Programmers are writing code at a higher level of abstraction: JavaScript, Python, Java, Ruby, and PHP are now the most popular languages on GitHub (in that order), surpassing lower-level languages such as C and C++ [51]. Developers also increasingly focus on application-specific logic, reusing existing libraries for general functionality when possible [19, 23, 40].

New programming paradigms are also liberating developers from the distraction of managing servers [18, 52, 54]. In particular, a proliferation of new serverless platforms [5, 6, 14, 17, 20, 39, 45] allow developers to construct applications as a set of handlers, called *lambdas*, commonly written in Python (or some other high-level language), that execute in response to events, such as web requests or data generation. Serverless providers automatically scale the number of handlers up and down to accommodate load so that developers need not worry about the number or configuration of machines serving their workload. Using serverless platforms is often very economical: billing granularity is in fractions of a second, and there is generally no tenant charge for idle time.

These three strategies (*i.e.*, programming at higher abstraction levels, reusing libraries, and decomposing applications into auto-scaling serverless lambdas) improve developer velocity, but they also create new infrastructure problems. Specifically, these techniques make process cold-start more expensive and frequent. Languages such as Python and JavaScript require heavy runtimes, making startup over 10× slower than launching an equivalent C program [1]. Reusing code introduces further startup latency from library loading and initialization [4, 8, 26, 27]. Serverless computing amplifies these costs: if a monolithic application is decomposed to N serverless lambdas, startup frequency is similarly amplified. Lambdas are typically isolated from each other via containers, which entail further sandboxing overheads [31].

Fast cold start is important for both tenants and providers. A graceful reaction to flash crowds [15, 22] requires concurrent low-latency deployment to many workers. From a provider perspective, avoiding cold starts can be quite costly. Most serverless platforms currently wait minutes or hours to recycle idle, unbilled lambda instances [50]. If cold start is made faster, providers will be able to reclaim idle resources and rebalance load across machines more aggressively.

In order to better understand the sandboxing and application characteristics that interfere with efficient cold start, we perform two detailed studies. First, we analyze the performance and scalability of various Linux isolation primitives. Among other findings, we uncover scalability bottlenecks in the network and mount namespaces and identify lighter-weight alternatives. Second, we study 876K Python projects from GitHub and analyzing 101K unique packages from the PyPI repository. We find that many popular packages take 100 ms to import, and installing them can take seconds. Although the entire 1.5 TB package set is too large to keep in memory, we find that 36% of imports are to just 0.02% of packages.

Based on these findings, we implement SOCK (roughly for serverless-optimized containers), a special-purpose container system with two goals: (1) *low-latency invocation* for Python handlers that import libraries and (2) *efficient sandbox initialization* so that individual workers can achieve high steady-state throughput. We integrate SOCK with the OpenLambda [20] serverless platform, replacing Docker as the primary sandboxing mechanism.

SOCK is based on three novel techniques. First, SOCK uses lightweight isolation primitives, avoiding the performance bottlenecks identified in our Linux primitive study, to achieve an 18× speedup over Docker. Second, SOCK provisions Python handlers using a generalized Zygote-provisioning strategy to avoid the Python initialization costs identified in our package study. In the simplest scenarios, this technique provides an additional 3× speedup by avoiding repeated initialization of the Python runtime. Third, we leverage our generalized Zygote mechanism to build a three-tiered package-aware caching system, achieving 45× speedups relative to SOCK containers without Zygote initialization. In an image-resizing case study, SOCK reduces cold-start platform overheads by 2.8× and 5.3× relative to AWS Lambda and OpenWhisk, respectively.

The rest of this paper is structured as follows. We study the costs of Linux provisioning primitives (§2) and application initialization (§3), and use these findings to guide the design and implementation of SOCK (§4). We then evaluate the performance of SOCK (§5), discuss related work (§6), and conclude (§7).

2. Deconstructing Container Performance

Serverless platforms often isolate lambdas with containers [14, 20, 39, 45]. Thus, optimizing container initialization is a key part of the lambda cold-start problem. In Linux, containerization is not a single-cohesive abstraction. Rather, general-purpose tools such as Docker [36] are commonly used to construct containers using a variety of Linux mechanisms to allocate storage, isolate resources logically, and isolate performance. The flexibility Linux provides also creates an opportunity to design a variety of special-purpose container systems. In this section, we hope to inform the design of SOCK and other special-purpose container systems by analyzing the performance characteristics of the relevant Linux abstractions. In particular, we ask *how can one maximize density of container file systems per machine? What is the cost of isolating each major resource with namespaces, and which resources must be isolated in a serverless environment?* And *how can the cost of repeatedly initializing cgroups to isolate performance be avoided?* We perform our analysis on an 8-core m510 machine [11] with the 4.13.0-37 Linux kernel.

2.1 Container Storage

Containers typically execute using a root file system other than the host's file system. This protects the host's data and provides a place for the container's unique dependencies to be installed. Provisioning a file system for a container is a two step procedure: (1) populate a subdirectory of the host's file system with data and code needed by the container, and (2) make the subdirectory the root of the new container, such that code in the container can no longer access other host data. We explore alternative mechanisms for these population and access-dropping steps.

Populating a directory by physical copying is prohibitively slow, so all practical techniques rely on logical copying. Docker typically uses union file systems (*e.g.*, AUFS) for this purpose; this provides a flexible layered-composition mechanism and gives running containers copy-on-write access over underlying data. A simpler alternative is bind mounting. Bind mounting makes the same directory visible at multiple locations; there is no copy-on-write capability, so data that must be protected should only be bind-mounted as read-only in a container. To compare binds to layered file systems, we repeatedly mount and unmount from many tasks in parallel. Figure 1 shows the result: at scale, bind mounting is about twice as fast as AUFS.

Once a subdirectory on the host file system has been populated for use as a container root, the setup process must switch roots and drop access to other host file data. Linux provides two primitives for modifying the file-system visible to a container. The older `chroot` operation simply turns a subdirectory of the original root file system into the new root file system. A newer mount-namespace abstraction enables more interesting transformations: an `unshare` call (with certain arguments) gives a container a new set of mount points, originally identical to the host's set. The container's mount points can then be modified with mount, unmount, and other calls. It is even possible to reorganize the mount namespace of a container such that the container may see file system Y mounted on file system X (the container's root) while the host may see X mounted on Y (the host's root). There are cases where this powerful abstraction can be quite helpful, but overusing mount namespace flexibility "may quickly lead to insanity," as the Linux manpages warn [32].

We measure the scalability of mount namespaces, with results shown in Figure 2. We have a variable number of long-lived mount namespaces persisting throughout the experiment (x-axis). We churn namespaces, concurrently creating and deleting them (concurrency is shown by different lines). We observe that churn performance scales poorly with the number of prior existing namespaces: as the number of host mounts grows large, the rate at which namespaces can be cloned approaches zero. We also evaluate `chroot` (not shown), and find that using it entails negligible overhead (`chroot` latency is $< 1\mu s$).

2.2 Logical Isolation: Namespace Primitives

We have already described how mount namespaces can be used to virtualize storage: multiple containers can have

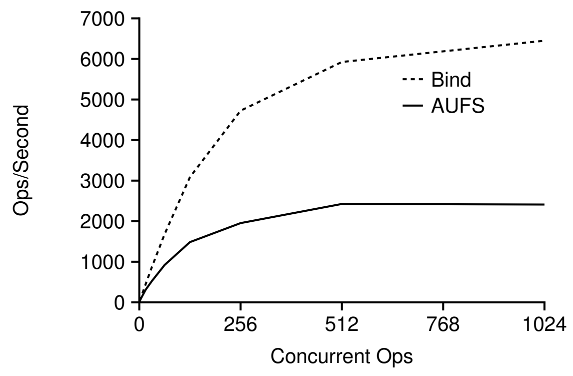

Figure 1. Storage Primitives. *The performance of mounting and unmounting AUFS file systems is compared to the performance of doing the same with bind mounts.*

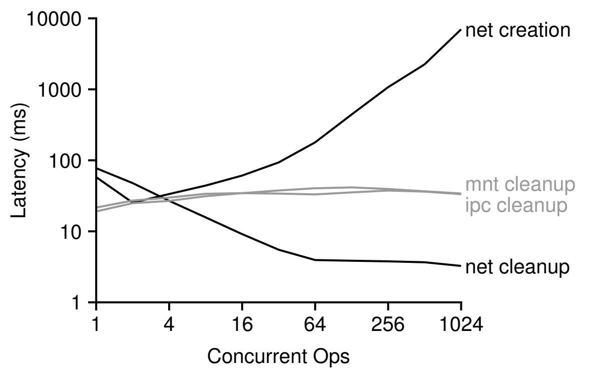

Figure 3. Namespace Primitives. *Lines show the latencies for copy_net_ns, put_mnt_ns and put_ipc_ns, and the average time spent by the network cleanup task per namespace.*

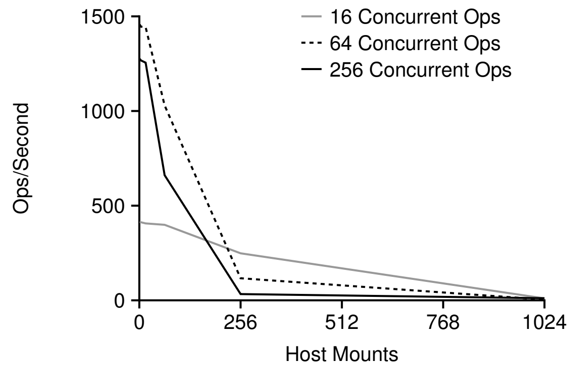

Figure 2. Mount Scalability. *This shows the rate at which processes can be unshared into new mount namespaces (y-axis) as the number of existing mounts varies (x-axis).*

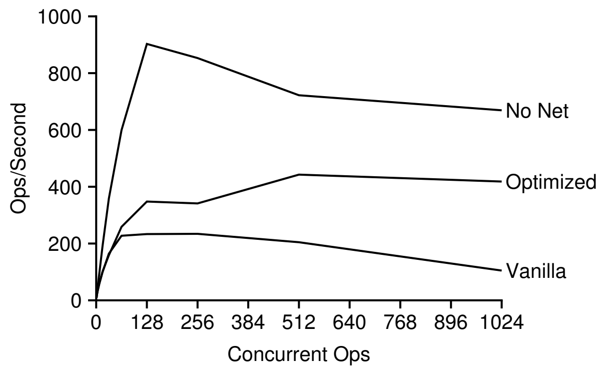

Figure 4. Network Namespace Performance. *A given number of containers (x-axis) are created and deleted in parallel with multiple network namespace configurations.*

access to their own virtual roots, backed by different physical directories in the host. Linux's network namespaces similarly 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). In this section, we study the collective use of mount and network namespaces, along with UTS, IPC, and PID namespaces [37] (user and cgroup namespaces are not evaluated here). The `unshare` 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.

We exercise namespace creation and cleanup performance by concurrently invoking `unshare` and exiting from a variable number of tasks. We instrument the kernel with `ftrace` to track where time is going. Figure 3 shows the latency of the four most expensive namespace operations (other latencies not shown were relatively insignificant). We observe that mount and IPC namespace cleanup entails latencies in the tens of milliseconds. Upon inspection of the kernel code, we found that both operations are waiting for an RCU grace period [35]. During this time, no global locks are held and no compute is consumed, so these latencies are relatively harmless to overall throughput; as observed earlier (§2.1), it is possible to create ∼1500 mount namespaces per second, as long as churn keeps the number of namespaces small over time.

Network namespaces are more problematic for both creation and cleanup due to a single global lock that is shared across network namespaces [13]. During creation, Linux iterates over all existing namespaces while holding the lock, searching for namespaces that should be notified of the configuration change; thus, costs increase proportionally as more namespaces are created. As with mount and IPC namespaces, network-namespace cleanup requires waiting for an RCU grace period. However, for network namespaces, a global lock is held during that period, creating a bottleneck. Fortunately, network namespaces are cleaned in batches, so the per-namespace cost becomes small at scale (as indicated by the downward-sloping "net cleanup" line).

Figure 4 shows the impact of network namespaces on overall creation/deletion throughput (*i.e.*, with all

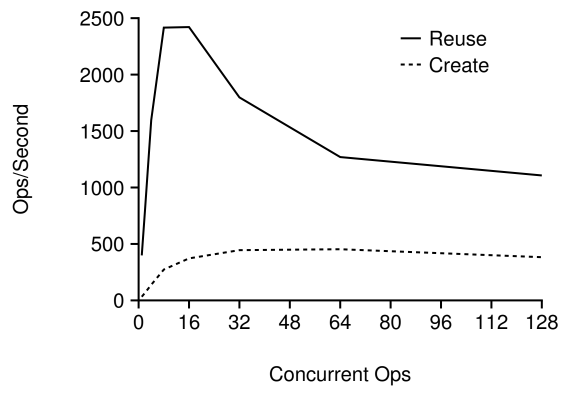

Figure 5. Cgroup Primitives. *Cgroup performance is shown for reuse and fresh-creation patterns.*

five namespaces). With unmodified network namespaces, throughput peaks at about 200 c/s (containers/second). With minor optimizations (disabling IPv6 and eliminating the costly broadcast code), it is possible to churn over 400 c/s. However, eliminating network namespaces entirely provides throughput of 900 c/s.

2.3 Performance Isolation: Cgroup Primitives

Linux provides performance isolation via the cgroup interface [9]. Processes may be assigned to cgroups, which are configured with limits on memory, CPU, file I/O, and other resources. Linux cgroups are easier to configure dynamically than namespaces. The API makes it simple to adjust the resource limits or reassign processes to different cgroups. In contrast, a mechanism for reassigning a process to a new PID namespace would need to overcome obstacles such as potential PID collisions.

The flexibility of cgroups makes two usage patterns viable. The first involves (1) creating a cgroup, (2) adding a process, (3) exiting the process, and (4) removing the cgroup; the second involves only Steps 2 and 3 (*i.e.*, the same cgroup is reused for different processes at different times). Figure 5 compares the cost of these two approaches while varying the numbers of tasks concurrently manipulating cgroups. Reusing is at least twice as fast as creating new cgroups each time. The best reuse performance is achieved with 16 threads (the number of CPU hyperthreads), suggesting cgroups do not suffer from the scaling issues we encountered with namespaces.

2.4 Serverless Implications

Our results have several implications for the design of serverless containers. First, in a serverless environment, all handlers run on one of a few base images, so the flexible stacking of union file systems may not be worth the the performance cost relative to bind mounts. Once a root location is created, file-system tree transformations that rely upon copying the mount namespace are costly at scale. When flexible file-system tree construction is not necessary, the cheaper `chroot` call may be used to drop access. Second, network namespaces are a major scalability bottleneck; while static port assignment may be useful in a server-based environment, serverless platforms such as AWS Lambda execute handlers behind a Network Address Translator [16], making network namespacing of little value. Third, reusing cgroups is twice as fast as creating new cgroups, suggesting that maintaining a pool of initialized cgroups may reduce startup latency and improve overall throughput.

3. Python Initialization Study

Even if lambdas are executed in lightweight sandboxes, language runtimes and package dependencies can make cold start slow [4, 8, 26, 27]. Many modern applications are accustomed to low-latency requests. For example, most Gmail remote-procedure calls are short, completing in under 100 ms (including Internet round trip) [20]. Of the short requests, the average latency is 27 ms, about the time it takes to start a Python interpreter and print a "hello world" message. Unless serverless platforms provide language- and library-specific cold-start optimizations, it will not be practical to decompose such applications into independently scaling lambdas. In this section, we analyze the performance cost of using popular Python libraries and evaluate the feasibility of optimizing initialization with caching. We ask: *what types of packages are most popular in Python applications? What are the initialization costs associated with using these packages?* And *how feasible is it to cache a large portion of mainstream package repositories on local lambda workers?*

3.1 Python Applications

We now consider the types of packages that future lambda applications might be likely to use, assuming efficient platform support. We scrape 876K Python projects from GitHub and extract likely dependencies on packages in the popular Python Package Index (PyPI) repository, resolving naming ambiguity in favor of more popular packages. 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.

Figure 6 shows the popularity of 20 packages that are most common as GitHub project dependencies. Skew is high: 36% of imports are to just 20 packages (0.02% of the packages in PyPI). The 20 packages roughly fall into five categories: web frameworks, analysis, communication, storage, and development. Many of these use cases are likely applicable to future serverless applications. Current web frameworks will likely need to be replaced by serverless-oriented frameworks, but compute-intense analysis is ideal for lambdas [21]. Many lambdas will need libraries for communicating with other services and for storing data externally [16]. Development

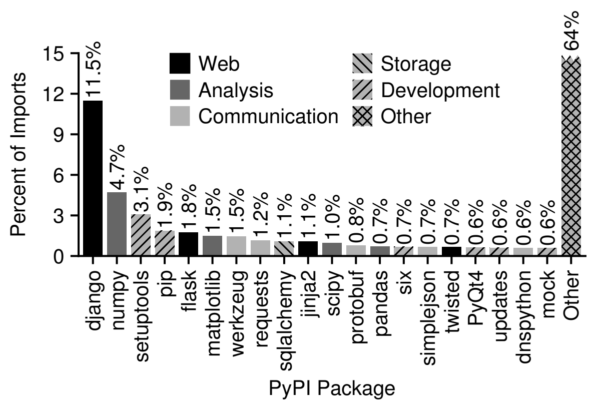

Figure 6. Package Popularity. *The twenty most used PyPI packages are shown. The bar labels represent the percentage of all GitHub-to-PyPI dependencies.*

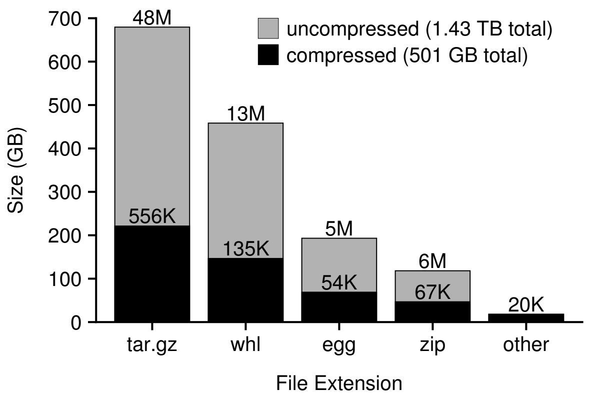

Figure 8. PyPI Package Data. *The size of the PyPI repository is shown, compressed and uncompressed, by file type (as of Mar 31, 2017). Bar labels show file counts.*

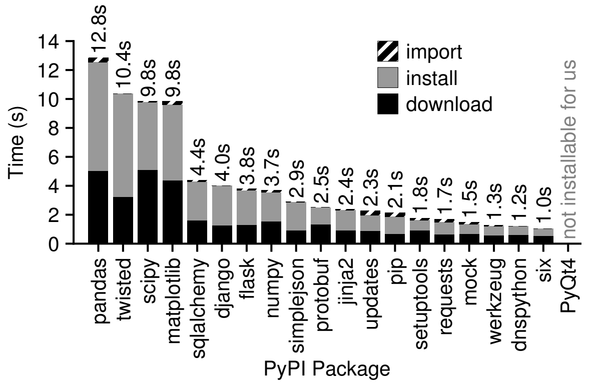

Figure 7. Startup Costs. *The download, install, and import times are shown for 20 popular Python packages, ordered by total initialization time.*

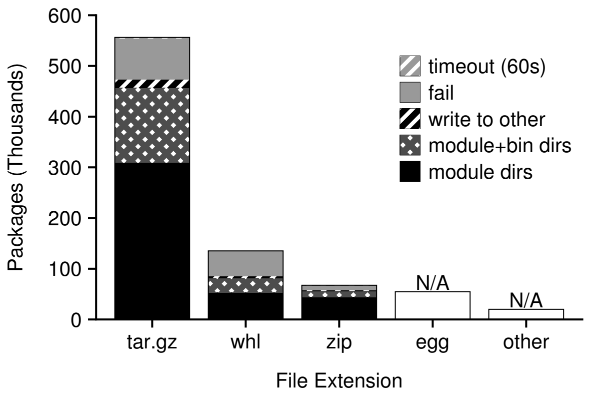

Figure 9. File-System Modifications. *The bars breakdown installations by the types of writes to the file system. The egg and other files can be used without extraction.*

libraries may be somewhat less relevant, but lambda-based parallel unit testing is an interesting use case.

If a package is being used for the first time, 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 7 shows these costs for each of the popular 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.

3.2 PyPI Repository

We now explore the feasibility of supporting full language repositories locally on serverless worker machines. We mirror and analyze the entire PyPI repository, which contains 101K unique packages. Figure 8 shows the footprint of the entire repository, including every version of every package, but excluding indexing files. The packages are about 1.5 TB total, or ∼0.5 TB compressed. Most packages are compressed as *.tar.gz* files or a zip-based format (*.whl*, *.egg*, or *.zip*). Across all format types, the average package contains about 100 files (*e.g.*, 135K *.whl* packages hold 13M compressed files).

We wish to understand how many of the PyPI packages could coexist when installed together. PyPI packages that unpack to a single directory can easily coexist with other installed packages, whereas packages that modify shared files may break other packages. We attempt to install every version of every PyPI package in its own Docker Ubuntu container (using a 1-minute timeout) and identify file creations and modifications. We ignore changes to temporary locations. Figure 9 shows the results for *.tar.gz*, *.whl*, and *.zip* distributions (*.egg* libraries are used directly without a prior installation, so we skip those). While fewer than 1% timed out, 18% simply failed to install in our container. 66% of succeeding installs only populate the local Python module directory (the *module dirs* category). Another 31% of succeeding installs modified just the module directories and the local bin directory (Python modules are sometimes bundled with various utilities). We conclude it is possible for 97% of installable packages to coexist in a single local install.

3.3 Serverless Implications

Downloading and installing a package and its dependencies from a local mirror takes seconds; furthermore, import of installed packages takes over 100 ms. Fortunately, our analysis indicates that storing large package repositories locally on disk is feasible. Strong popularity skew further creates opportunities to pre-import a subset of packages into interpreter memory [8].

4. SOCK with OpenLambda

In this section, we describe the design and implementation of SOCK, a container system optimized for use in serverless platforms. We integrate SOCK with the OpenLambda serverless platform, replacing Docker containers as the primary sandboxing mechanism for OpenLambda workers and using additional SOCK containers to implement Python package caching. We design SOCK to handle high-churn workloads at the worker level. The local churn may arise due to global workload changes, rebalancing, or aggressive reclamation of idle resources.

SOCK is based on two primary design goals. First, we want *low-latency invocation* for Python handlers that import libraries. Second, we want *efficient sandbox initialization* so that individual workers can achieve high steady-state throughput. A system that hides latency by maintaining pools of pre-initialized containers (*e.g.*, the LightVM approach [31]) would satisfy the first goal, but not the second. A system that could create many containers in parallel as part of a large batch might satisfy the second goal, but not the first. Satisfying both goals will make a serverless platform suitable for many applications and profitable for providers.

Our solution, SOCK, takes a three-pronged approach to satisfying these goals, based on our analysis of Linux containerization primitives (§2) and Python workloads (§3). First, we build a lean container system for sandboxing lambdas (§4.1). Second, we generalize Zygote provisioning to scale to large sets of untrusted packages (§4.2). Third, we design a three-layer caching system for reducing package install and import costs (§4.3).

4.1 Lean Containers

SOCK creates lean containers for lambdas by avoiding the expensive operations that are only necessary for general-purpose containers. Creating a container involves constructing a root file system, creating communication channels, and imposing isolation boundaries. Figure 10 illustrates SOCK's approach to these three tasks.

Storage: Provisioning container storage involves first populating a directory on the host to use as a container root. Bind mounting is faster using union file systems (§2.1), so SOCK uses bind mounts to stitch together a root from four host directories, indicated by the "F" label in Figure 10. Every container has the same Ubuntu base for its root file system ("base"); we can afford to back this by a RAM disk as every handler is required to use the same base. A packages directory used for package caching ("packages") is mounted over the base, as described later (§4.3). The same base and packages are read-only shared in every container. SOCK also binds handler code ("λ code") as read-only and a writable scratch directory ("scratch") in every container.

Once a directory has been populated as described, it should become the root directory. Tools such as Docker accomplish this by creating a new mount namespace, then restructuring it. We use the faster and simpler `chroot` operation (§2.1) since it is not necessary to selectively expose other host mounts within the container for serverless applications. SOCK containers always start with two processes ("init" and "helper" in Figure 10); both of these use `chroot` during container initialization, and any children launched from these processes inherit the same root.

Communication: The scratch-space mount of every SOCK container contains a Unix domain socket (the black pentagon in Figure 10) 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.

The channel is also used for a variety of control operations (§4.2). Some of these operations require privileged access to resources not normally accessible inside a container. Fortunately, the relevant resources (*i.e.*, namespaces and container roots) may be represented as file descriptors, which may be passed over Unix domain sockets. The manager can thus pass specific capabilities over the channel as necessary.

Isolation: Linux processes may be isolated with a combination of cgroup (for performance isolation) and namespace primitives (for logical isolation). It is relatively expensive to create cgroups; thus, OpenLambda creates a pool of cgroups (shown in Figure 10) that can be used upon SOCK container creation; cgroups are returned to the pool after container termination.

The "init" process is the first to run in a SOCK container; init creates a set of new namespaces with a call to `unshare`. The arguments to the call indicate that mount and network namespaces should not be used, because these were the two namespaces that scale poorly (§2.1 and §2.2). Mount namespaces are unnecessary because SOCK uses `chroot`. Network namespaces are unnecessary because requests arrive over Unix domain socket, not over a socket attached to a fixed port number, so port virtualization is not required.

4.2 Generalized Zygotes

Zygote provisioning is a technique where new processes are started as forks of an initial process, the Zygote,

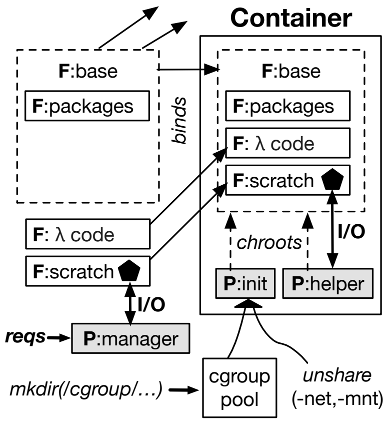

Figure 10. Lean Containers

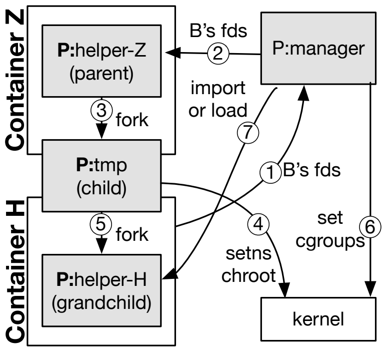

Figure 11. Generalized Zygotes

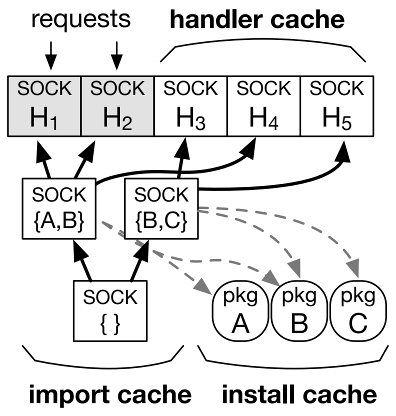

Figure 12. Serverless Caching

that has already pre-imported various libraries likely to be needed by applications, thereby saving child processes from repeatedly doing the same initialization work and consuming excess memory with multiple identical copies. Zygotes were first introduced on Android systems for Java applications [8]. 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.

As already described, SOCK containers start with two processes, an init process (responsible for setting up namespaces) and a helper process. The helper process is a Python program that listens on the SOCK communication channel; it is capable of (a) pre-importing modules and (b) loading lambda handlers to receive subsequent forwarded events. These two capabilities are the basis for a simple Zygote mechanism. A *Zygote helper* first pre-imports a set of modules. Then, when a lambda is invoked requiring those modules, the Zygote helper is forked to quickly create a new *handler helper*, which then loads the lambda code to handle a forwarded request.

We assume packages that may be pre-imported may be malicious [48], and handlers certainly may be malicious, so both Zygote helpers and handler helpers must run in containers. The key challenge is using Linux APIs such that the forked process lands in a new container, distinct from the container housing the Zygote helper.

Figure 11 illustrates how the SOCK protocol provisions a helper handler ("helper-H" in "Container H") from a helper Zygote ("helper-Z" in "Container Z"). **(1)** The manager obtains references, represented as file descriptors (fds), to the namespaces and the root file system of the new container. **(2)** The fds are passed to helper-Z, which **(3)** forks a child process, "tmp". **(4)** The child then changes roots to the new container with a combination of `fchdir(fd)` and `chroot(".")` calls. The child also calls `setns` (set namespace) for each namespace to relocate to the new container. **(5)** One peculiarity of `setns` is that after the call, the relocation has only partially been applied to all namespaces for the caller. Thus, the child calls fork again, creating a grandchild helper ("helper-H" in the figure) that executes fully in the new container with respect to namespaces. **(6)** The manager then moves the grandchild to the new cgroup. **(7)** Finally, the helper listens on the channel for the next commands; the manager will direct the helper to load the lambda code, and will then forward a request to the lambda.

The above protocol describes how SOCK provisions a handler container from a Zygote container. When OpenLambda starts, a single Zygote that imports no modules is always provisioned. In order to benefit from pre-importing modules, SOCK can create additional Zygotes that import various module subsets. Except for the first Zygote, new Zygotes are provisioned from existing Zygotes. The protocol for provisioning a new Zygote container is identical to the protocol for provisioning a new handler container, except for the final step 7. Instead of loading handler code and processing requests, a new Zygote pre-imports a specified list of modules, then waits to be used for the provisioning of other containers.

Provisioning handlers from Zygotes and creating new Zygotes from other Zygotes means that all the interpreters form a tree, with copy-on-write memory unbroken by any call to `exec`. This sharing of physical pages between processes reduces memory consumption [2]. Initialization of the Python runtime and packages will only be done once, and subsequent initialization will be faster.

If a module loaded by a Zygote is malicious, it may interfere with the provisioning protocol (*e.g.*, by modifying the helper protocol so that calls to `setns` are skipped). Fortunately, the Zygote is sandboxed in a container, and

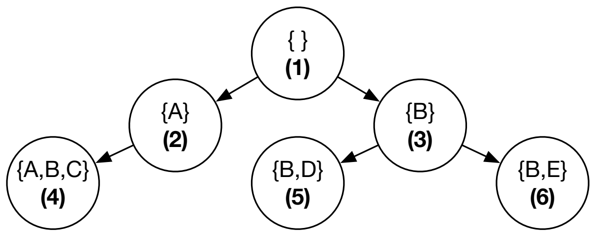

Figure 13. Tree Cache. *Numbered circles represent Zygotes in the cache, and sets of letters indicate the packages imported by a process. Arrows represent the parent-child relationships between interpreter processes.*

will never be passed descriptors referring to unrelated containers, so a malicious process cannot escape into arbitrary containers or the host. SOCK protects innocent lambdas by never initializing them from a Zygote that has pre-imported modules not required by the lambda.

4.3 Serverless Caching

We use SOCK to build a three-tier caching system, shown in Figure 12. First, a *handler cache* maintains idle handler containers in a paused state; the same approach is taken by AWS Lambda [49]. 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.

Second, an *install cache* contains a large, static set of pre-installed packages on disk. Our measurements show that 97% of installable PyPI packages could coexist in such a installation. 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 (§4.2). 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. Import caching entails new decisions for handling hits. In traditional caches, lookup results in a simple hit or miss; in contrast, SOCK always hits at least one cache entry and often must decide between alternative Zygotes. Eviction is also complicated by copy-on-write sharing of memory pages between Zygotes, which obfuscates the consumption of individuals. We now describe SOCK's selection and eviction policies.

Import-Cache Selection: Suppose (in the context of Figure 13) that a handler is invoked that requires packages A and B. Entry 4 is a tempting choice to use as the template for our new interpreter; it would provide the best performance because all requisite packages are already imported. However, if package C is malicious, we expose the handler to code that it did not voluntarily import. We could potentially vet a subset of packages to be deemed safe, but we should generally not use cache entries that pre-import packages not requested by a handler. This leaves cache Entries 2 and 3 as reasonable candidates. The import cache decides between such alternatives by choosing the entry with the most matching packages, breaking ties randomly. When SOCK must use an entry X that is not an exact match, it first replicates X to a new entry Y, imports the remaining packages in Y, and finally replicates from Y to provision for the handler.

Import-Cache Eviction: The import cache measures the cumulative memory utilization of all entries; when utilization surpasses a limit, a background process begins evicting entries. Deciding which interpreters to evict is challenging because the shared memory between interpreters makes it difficult to account for the memory used by a particular entry. The import cache relies on a simple runtime model to estimate potential memory reclamation; the model identifies the packages included by an interpreter that are not included by the parent entry. The model uses the on-disk size of the packages as a heuristic for estimating memory cost. The import cache treats the sum of these sizes as the *benefit* of eviction and the number of uses over a recent time interval as the *cost* of eviction, evicting the entry with highest benefit-to-cost ratio.

5. Evaluation

We now evaluate the performance of SOCK relative to Docker-based OpenLambda and other platforms. We run experiments on two m510 machines [11] with the 4.13.0-37 Linux kernel: a package mirror and an OpenLambda worker. The machines have 8-core 2.0 GHz Xeon D-1548 processors, 64 GB of RAM, and a 256 GB NVMe SSD. We allocate 5 GB of memory for the handler cache and 25 GB for the import cache. We consider the following questions: *What speedups do SOCK containers provide OpenLambda* (§5.1)? *Does built-in package support reduce cold-start latency for applications with dependencies* (§5.2)? *How does SOCK scale with the number of lambdas and packages* (§5.3)? And *how does SOCK compare to other platforms for a real workload* (§5.4)?

5.1 Container Optimizations

SOCK avoids many of the expensive operations necessary to construct a general-purpose container (*e.g.*, network namespaces, layered file systems, and fresh cgroups). 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. Figure 14 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 re-

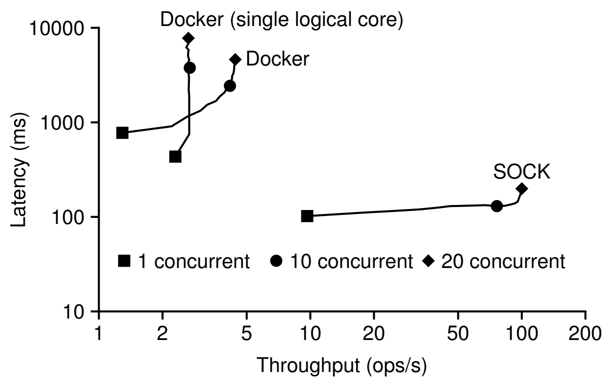

Figure 14. Docker vs. SOCK. *Request throughput (x-axis) and latency (y-axis) are shown for SOCK (without Zygotes) and Docker for varying concurrency.*

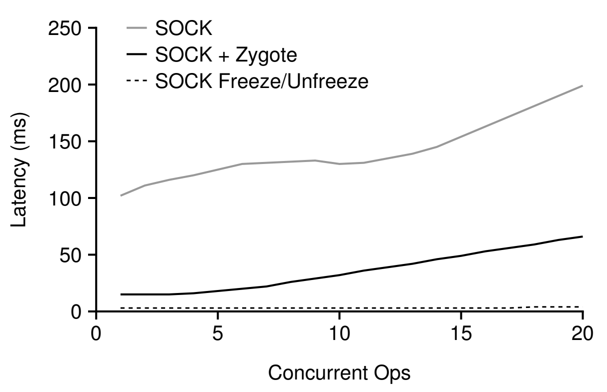

Figure 16. Container Reuse vs. Fast Creation. *SOCK container creation is compared to the freeze/unfreeze operation that occurs when there are repeated calls to the same lambda.*

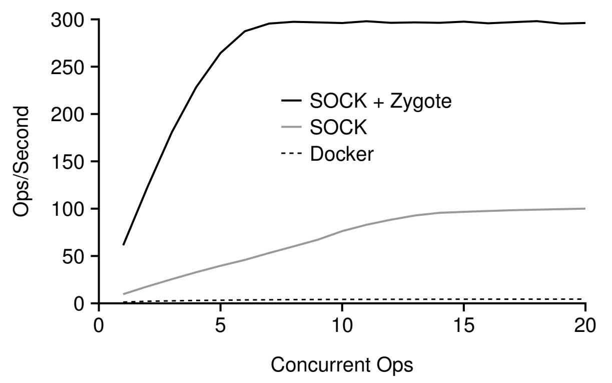

Figure 15. Interpreter Preinitialization. *HTTP Request Throughput is shown relative to number of concurrent requests.*

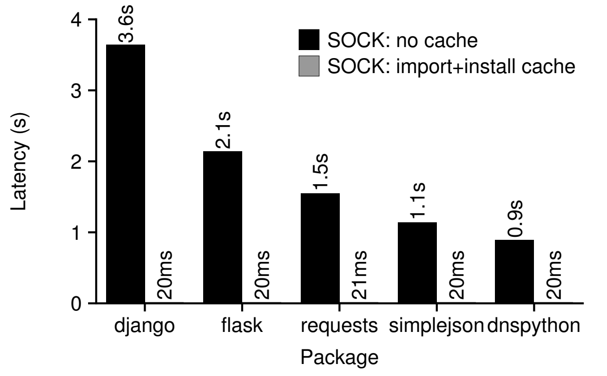

Figure 17. Pre-Imported Packages. *SOCK latency with and without package caches are shown.*

quests, SOCK has a throughput of 76 requests/second (18× faster than Docker) with an average latency of 130 milliseconds (19× faster). Some of the namespaces used by Docker rely heavily on RCUs (§2.2), which scale poorly with the number of cores [34]. Figure 14 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.

SOCK also improves performance by using Zygote-style preinitialization. Even if a lambda uses no libraries, provisioning a runtime by forking an existing Python interpreter is faster than starting from scratch. Figure 15 compares SOCK throughput with and without Zygote preinitialization. Using Zygotes provides SOCK with an additional 3× throughput improvement at scale.

OpenLambda, like AWS Lambda [49], keeps recently used handlers that are idle in a paused state in order to avoid cold start should another request arrive. We now compare the latency of SOCK cold start to the latency of unpause, as shown in Figure 16. Although Zygotes have reduced no-op cold-start latency to 32 ms ($concurrency = 10$), unpausing takes only 3 ms. Although SOCK cold-start optimizations enable more aggressive resource reclamation, it is still beneficial to pause idle handlers before immediately evicting them.

5.2 Package Optimizations

SOCK provides two package-oriented optimizations. First, SOCK generalizes the Zygote approach so that new containers can be allocated by one of many different Zygote containers, each with different packages pre-imported, based on the current workload (import caching). Second, a large subset of packages are pre-installed to a partition that is bind-mounted read-only in every container (install caching).

We first evaluate these optimizations together with a simple workload, where a single task sequentially invokes different lambdas that use the same single library, but perform no work. Figure 17 shows the result. Without optimizations, downloading, installing, and importing usually takes at least a second. The optimizations reduce latency to 20 ms, at least a 45× improvement.

To better understand the contributions of the three caching layers (*i.e.*, the new import and install caches and the old handler cache), we repeat the experiment in Figure 17 for django, evaluating all caches, no caches, and

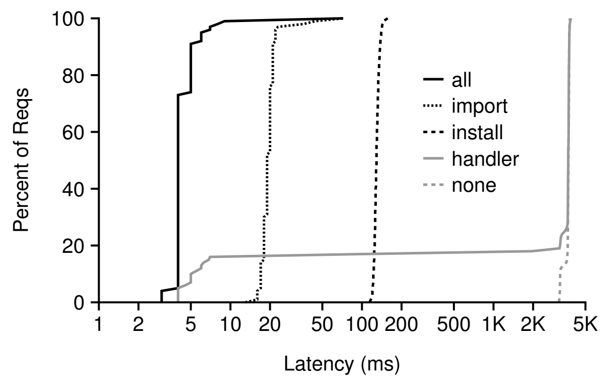

Figure 18. Individual Caches. *Each line shows a latency CDF for a different configuration for the django experiment.*

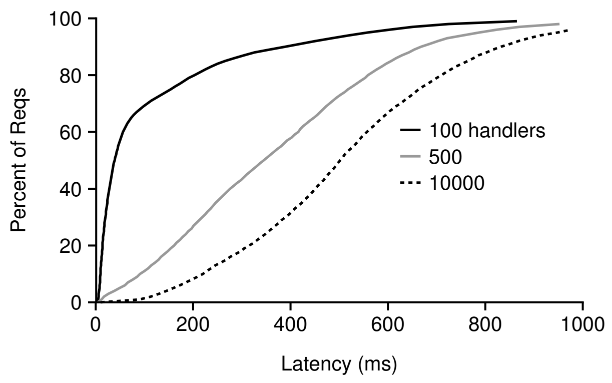

Figure 20. Scalability: Synthetic Packages. *Each line represents a CDF of latencies for a different working-set size.*

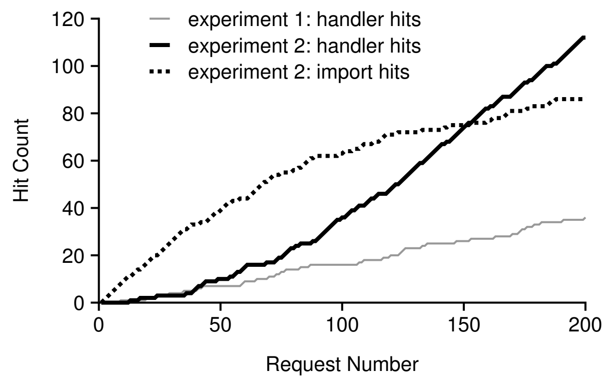

Figure 19. Handler/Import Cache Interactions. *Hit counts are shown during cache warmup (the first 200 requests) for two cache configurations.*

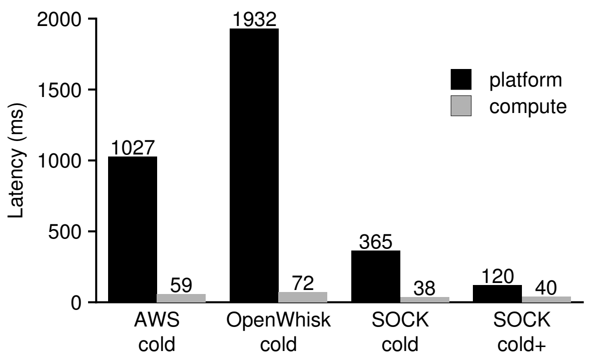

Figure 21. AWS Lambda and OpenWhisk. *Platform and compute costs are shown for cold requests to an image-resizing lambda. S3 latencies are excluded to minimize noise.*

each cache in isolation. For each experiment, 100 different lambdas import django, and a single task sequentially invokes randomly-chosen lambdas. Figure 18 shows the results. The handler cache has bimodal latency: it usually misses, but is fastest upon a hit. The working set fits in the import cache, which provides consistent latencies around 20 ms; the install cache is also consistent, but slower. Using all caches together provides better performance than any one individually.

When import caching is enabled, processes in the handler cache and processes in the import cache are part of the same process tree. This structure leads to deduplication: multiple processes in the handler cache can share the same memory page on a copy-on-write basis with a parent process in the import cache. This allows the handler cache to maintain more cache entries. Figure 19 illustrates this helpful interaction. We issue 200 requests to many different lambdas, all of which import django, without an import cache (experiment 1) and with an import cache (experiment 2). In the first experiment, the handler cache has 18% hits. In the second, deduplication allows the handler cache to maintain more entries, achieving 56% hits.

5.3 Scalability

We stress test SOCK with a large set of artificial packages (100K). The packages generate CPU load and memory load, similar to measured overheads of 20 popular packages (§3.1). We create dependencies between packages similar to the PyPI dependency structure. Each handler imports 1-3 packages directly. The packages used are decided randomly based on package popularity; popularity is randomly assigned with a Zipfian distribution, $s = 2.5$. All packages are pre-installed to the install cache.

We also vary the number of handlers (100 to 10K). A small working set exercises the handler cache, and a large working set exercises the install cache. The import cache should service mid-sized working sets. Handlers are executed uniformly at random as fast as possible by 10 concurrent tasks. Figure 20 shows a latency CDF for each working set size. With 100 handlers, SOCK achieves low latency (39 ms median). For 10K handlers, 88% percent of requests must be serviced from the install cache, so the median latency is 502 ms. For 500 handlers, the import cache absorbs 46% of the load, and the handler cache absorbs 6.4%, resulting in 345 ms latencies.

5.4 Case Study: Image Resizing

In order to evaluate a real serverless application, we implement on-demand image resizing [28]. A lambda reads an image from AWS S3, uses the Pillow package to resize it [10], and writes the output back to AWS S3. For this experiment, we compare SOCK to AWS Lambda and OpenWhisk, 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. We use AWS's US East region for EC2, Lambda, and S3.

For SOCK, we preinstall Pillow and the AWS SDK [44] (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 21 show compute and platform results for each platform (average of 50 runs). "SOCK cold" has a platform latency of 365 ms, 2.8× faster than AWS Lambda and 5.3× 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 3× to 120 ms.

6. Related Work

Since the introduction of AWS Lambda in 2014 [5], many new serverless platforms have become available [6, 14, 17, 39, 45]. We build SOCK over OpenLambda [20]. SOCK implements and extends our earlier Pipsqueak proposal for efficient package initialization [38].

In this work, we benchmark various task-provisioning primitives and measure package initialization costs. Prior studies have ported various applications to the lambda model in order to evaluate platform performance [21, 33]. Spillner *et al.* [46] ported Java applications to AWS Lambda to compare performance against other platforms, and Fouladi *et al.* [16] built a video encoding platform over lambdas. Wang *et al.* [50] reverse engineer many design decisions made by major serverless platforms.

There has been a recent revival of interest in sandboxing technologies. Linux containers, made popular through Docker [36], represent the most mainstream technology; as we show herein, the state of the art is not yet tuned to support lambda workloads. OpenWhisk, which uses Docker containers, hides latency by maintaining pools of ready containers [47]. Recent alternatives to traditional containerization are based on library operating systems, enclaves, and unikernels [7, 24, 29, 31, 41, 42].

The SOCK import cache is a generalization of the Zygote approach first used by Android [8] for Java processes. Akkus *et al.* [1] also leverage this technique to efficiently launch multiple lambdas in the same container when the lambdas belong to the same application. Zygotes have also been used for browser processes (sometimes in conjunction with namespaces [12]). We believe SOCK's generalized Zygote strategy should be generally applicable to other language runtimes that dynamically load libraries or have other initialization costs such as JIT-compilation (*e.g.*, the v8 engine for Node.js [25] or the CLR runtime for C# [3, 30]); however, it is not obvious how SOCK techniques could be applied to statically-linked applications (*e.g.*, most Go programs [53]).

Process caching often has security implications. For example, HotTub [27] reuses Java interpreters, but not between different Linux users. Although the Zygote approach allows cache sharing between users, Lee *et al.* [26] observed that forking many child processes from the same parent without calling `exec` undermines address-space randomization; their solution was Morula, a system that runs `exec` every time, but maintains a pool of preinitialized interpreters; this approach trades overall system throughput for randomization.

7. 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 identify container initialization and package dependencies as common causes of slow lambda startup. Based on our analysis, we build SOCK, a streamlined container system optimized for serverless workloads that avoids major kernel bottlenecks. We further generalize Zygote provisioning and build 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.

Acknowledgements

We thank the anonymous ATC reviewers, 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] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. SAND: Towards High-Performance Serverless Computing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, 2018.

[2] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 0.91 edition, May 2015.

[3] Assemblies in the Common Language Runtime. https://docs.microsoft.com/en-us/dotnet/framework/app-domains/assemblies-in-the-common-language-runtime, May 2018.

[4] AWS Developer Forums: Java Lambda Inappropriate for Quick Calls? https://forums.aws.amazon.com/thread.jspa?messageID=679050, July 2015.

[5] AWS Lambda. https://aws.amazon.com/lambda/, February 2018.

[6] Microsoft Azure Functions. https://azure.microsoft.com/en-us/services/functions/, February 2018.

[7] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding Applications from an Untrusted Cloud with Haven. *ACM Transactions on Computer Systems (TOCS)*, 33(3):8, 2015.

[8] Dan Bornstein. Dalvik Virtual Machine Internals. Talk at Google I/O, 2008.

[9] cgroups - Linux Control Groups. http://man7.org/linux/man-pages/man7/cgroups.7.html, May 2018.

[10] Alex Clark. Pillow Python Imaging Library. https://pillow.readthedocs.io/en/latest/, February 2018.

[11] CloudLab. https://www.cloudlab.us/, February 2018.

[12] Chromium Docs. Linux Sandboxing. https://chromium.googlesource.com/chromium/src/+/lkcr/docs/linux_sandboxing.md, February 2018.

[13] Eric Dumazet. Re: net: cleanup_net is slow. https://lkml.org/lkml/2017/4/21/533, April 2017.

[14] Alex Ellis. Introducing Functions as a Service (OpenFaaS). https://blog.alexellis.io/introducing-functions-as-a-service/, August 2017.

[15] Jeremy Elson and Jon Howell. Handling Flash Crowds from Your Garage. In *USENIX 2008 Annual Technical Conference*, ATC'08, pages 171–184. USENIX Association, 2008.

[16] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 363–376, Boston, MA, 2017. USENIX Association.

[17] Google Cloud Functions. https://cloud.google.com/functions/docs/, February 2018.

[18] Jim Gray. Why Do Computers Stop and What Can We Do About It? In *6th International Conference on Reliability and Distributed Databases*, June 1987.

[19] Tyler Harter, Chris Dragga, Michael Vaughn, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. A File is Not a File: Understanding the I/O Behavior of Apple Desktop Applications. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 71–83. ACM, 2011.

[20] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Serverless Computation with OpenLambda. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*, Denver, CO, 2016. USENIX Association.

[21] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. Occupy the Cloud: Distributed Computing for the 99%. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 445–451. ACM, 2017.

[22] Jaeyeon Jung, Balachander Krishnamurthy, and Michael Rabinovich. Flash Crowds and Denial of Service Attacks: Characterization and Implications for CDNs and Web Sites. In *Proceedings of the 11th International Conference on World Wide Web*, WWW '02, pages 293–304. ACM, 2002.

[23] Hyojun Kim, Nitin Agrawal, and Cristian Ungureanu. Revisiting Storage for Smartphones. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST'12, pages 17–17. USENIX Association, 2012.

[24] Nicolas Lacasse. Open-Sourcing gVisor, a Sandboxed Container Runtime. https://cloudplatform.googleblog.com/2018/05/Open-sourcing-gVisor-a-sandboxed-container-runtime.html, May 2018.

[25] Thibault Laurens. How the V8 Engine Works? http://thibaultlaurens.github.io/javascript/2013/04/29/how-the-v8-engine-works/, April 2013.

[26] Byoungyoung Lee, Long Lu, Tielei Wang, Taesoo Kim, and Wenke Lee. From Zygote to Morula: Fortifying Weakened ASLR on Android. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 424–439. IEEE, 2014.

[27] David Lion, Adrian Chiu, Hailong Sun, Xin Zhuang, Nikola Grcevski, and Ding Yuan. Don't Get Caught In the Cold, Warm-up Your JVM. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI '16)*, Savannah, Georgia, October 2016.

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

[29] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Uniker-

nels: Library Operating Systems for the Cloud. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*, Houston, Texas, March 2013.

[30] Managed Execution Process. https://docs.microsoft.com/en-us/dotnet/standard/managed-execution-process, May 2018.

[31] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. My VM is Lighter (and Safer) Than Your Container. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 218–233. ACM, 2017.

[32] Linux Manpages. pivot_root (2) - Linux Man Pages. https://www.systutorials.com/docs/linux/man/2-pivot_root/, February 2018.

[33] Garrett McGrath and Paul R. Brenner. Serverless Computing: Design, Implementation, and Performance. In *Distributed Computing Systems Workshops (ICDCSW), 2017 IEEE 37th International Conference on*, pages 405–410. IEEE, 2017.

[34] Paul E McKenney. Introduction to RCU Concepts: Liberal application of procrastination for accommodation of the laws of physics for more than two decades! In *LinuxCon Europe 2013*, October 2013.

[35] Paul E McKenney, Silas Boyd-Wickizer, and Jonathan Walpole. RCU Usage in the Linux Kernel: One Decade Later, 2013.

[36] Dirk Merkel. Docker: Lightweight Linux Containers for Consistent Development and Deployment. Linux Journal, Issue 239, March 2014.

[37] namespaces(7) - overview of Linux namespaces. http://man7.org/linux/man-pages/man7/namespaces.7.html, May 2018.

[38] Edward Oakes, Leon Yang, Kevin Houck, Tyler Harter, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Pipsqueak: Lean Lambdas with Large Libraries. In *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pages 395–400, June 2017.

[39] IBM Cloud Functions. https://www.ibm.com/cloud/functions, February 2018.

[40] Rob Pike. Another Go at Language Design. http://www.stanford.edu/class/ee380/Abstracts/100428.html, April 2010.

[41] Donald E. Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C. Hunt. Rethinking the Library OS from the Top Down. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 291–304. ACM, 2011.

[42] Nelly Porter, Jason Garms, and Sergey Simakov. Introducing Asylo: an Open-Source Framework for Confidential Computing. https://cloudplatform.googleblog.com/2018/05/Introducing-Asylo-an-open-source-framework-for-confidential-computing.html, May 2018.

[43] Eric Ries. *The Lean Startup*. Crown Business, September 2011.

[44] Amazon Web Services. The AWS SDK for Python. https://boto3.readthedocs.io/en/latest/, February 2018.

[45] Shaun Smith. Announcing Fn: An Open Source Serverless Functions Platform. https://blogs.oracle.com/developers/announcing-fn, October 2017.

[46] Josef Spillner and Serhii Dorodko. Java Code Analysis and Transformation into AWS Lambda Functions. *CoRR*, abs/1702.05510, 2017.

[47] Markus Thömmes. Squeezing the milliseconds: How to make serverless platforms blazing fast! https://medium.com/openwhisk/squeezing-the-milliseconds-how-to-make-serverless-platforms-blazing-fast-aea0e9951bd0, April 2017.

[48] Nikolai Philipp Tschacher. Typosquatting in Programming Language Package Managers. Bachelor Thesis, University of Hamburg (6632193). http://incolumitas.com/data/thesis.pdf, March 2016.

[49] Tim Wagner. Understanding Container Reuse in AWS Lambda. https://aws.amazon.com/blogs/compute/container-reuse-in-lambda/, December 2014.

[50] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. Peeking Behind the Curtains of Serverless Platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, 2018.

[51] Matt Weinberger. The 15 Most Popular Programming Languages, According to the 'Facebook for Programmers'. http://www.businessinsider.com/the-9-most-popular-programming-languages-according-to-the-facebook-for-programmers-2017-10#1-javascript-15, October 2017.

[52] Andrew Whitaker, Richard S. Cox, and Steven D. Gribble. Configuration Debugging As Search: Finding the Needle in the Haystack. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 6–6. USENIX Association, 2004.

[53] Matt Williams. Go executables are statically linked, except when they are not. http://matthewkwilliams.com/index.php/2014/09/28/go-executables-are-statically-linked-except-when-they-are-not/, September 2014.

[54] Bowei Xi, Zhen Liu, Mukund Raghavachari, Cathy H. Xia, and Li Zhang. A Smart Hill-Climbing Algorithm for Application Server Configuration. In *Proceedings of the 13th international conference on World Wide Web*, pages 287–296. ACM, 2004.