USETL: Unikernels for Serverless Extract Transform and Load
Why should you settle for less?

Henrique Fingler
The University of Texas at Austin
hfingler@cs.utexas.edu

Amogh Akshintala
University of North Carolina at Chapel Hill
aakshintala@cs.unc.edu

Christopher J. Rossbach
The University of Texas at Austin
VMware Research Group
rossbach@cs.utexas.edu

Abstract

Growing popularity of serverless functions is driving the need to optimize the execution platform to reduce resource usage and increase the number of functions that can be executed concurrently. This reduces the provider's costs and increases profit. While current serverless solutions use containers and/or virtual machines, we propose a unikernel based design called *USETL* which is specialized for serverless *extract, transform, load* (ETL) workloads. Our design is motivated by a number of key observations: serverless functions are stateless, ephemeral and event-driven. Further, each function's specific purpose is known at invocation time. Unikernels are a natural fit for execution contexts with these properties: they are minimal kernels packaged with a single application in a single address space, which makes them incredibly lightweight. Our design removes network and storage components entirely, replacing them with high level APIs tailored to the needs of serverless ETL functions. Virtualizing I/O at the runtime library interface reduces memory and CPU overheads, yielding higher consolidation density.

CCS Concepts • **Security and privacy** → **Virtualization and security**; • **Software and its engineering** → *Operating systems*; • **Computer systems organization** → *Cloud computing*.

ACM Reference Format:
Henrique Fingler, Amogh Akshintala, and Christopher J. Rossbach. 2019. *USETL*: Unikernels for Serverless Extract Transform and Load *Why should you settle for less?*. In *10th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys '19), August 19–20, 2019, Hangzhou, China.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3343737.3343750

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
APSys '19, August 19–20, 2019, Hangzhou, China
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6893-3/19/08…$15.00
https://doi.org/10.1145/3343737.3343750

1 Introduction

Serverless functions - or Functions as a Service (FaaS) - were among the fastest growing cloud services in 2018 [21]. The appeal of serverless computing is clear: users are billed only for function execution time and complex distributed systems problems such as elasticity and infrastructure management are relegated to the service provider. Most cloud providers offer a serverless platform: e.g., AWS Lambda [4], Google Functions [6], IBM Cloud Functions [7], and Azure Functions [5] to name a few. Improving the generality and efficiency of serverless infrastructure is an active area of research [24, 25, 28, 32, 37, 42, 45].

Each cloud provider takes a different approach to guaranteeing elasticity, isolation, and performance in their serverless platform. Unfortunately, none of these approaches are optimal. Let us consider 2 examples: Google uses a sandboxed container runtime called gVisor [11] while AWS uses a combination of virtual machines and containers [10]. gVisor interposes system calls in the container, and translates them into a limited set of system calls to the host [27]. The host OS is therefore shared by mutually distrustful tenants, which opens the door for a wide range of exploits. For example, arbitrary host files could be overwritten [1], privileged memory could be read and written by an unprivileged container [16]. In some cases, this virtualization approach can lead to a larger attack surface: for some applications, the number of unique system calls gVisor makes to the host is greater than what a regular Docker runtime would make [13].

AWS, on the other hand, isolates each cloud tenant in VMs. Functions are executed in containers inside these VMs. Predictably, this scheme increases overhead: each VM has its own kernel, increasing memory consumption, as cross-VM page deduplication is rare, even when all the VMs are running the same OS [26]. The number of virtualization layers that an application must cross also adds latency overhead. Consider, for example, a function sending a network message. The message must go through the container's runtime security policies (e.g. `seccomp` filters), traverse the guest kernel network stack, be sent through a virtual device and be relayed between the host's kernel and user mode at least once per message. If the network was set up using tap devices, which is what KVM's documentation [14] suggests

for public bridging, the message will go through the kernel mode twice and through the user mode once before it is sent to the external network. [1] Further, each VM has its own guest OS scheduler, in addition to the host OS scheduler, leading to well-known double scheduling pathologies [30, 43, 46, 48].

In order to get both the isolation guarantees of VMs and the flexibility and agility of containers, we explore the use of unikernels. Unikernels [38] consist of a minimal operating system kernel and a single application packaged (or *baked*) with it, in a single address space. Unikernels can typically run on either bare-metal or on a hypervisor.

To efficiently support serverless functions, we propose to specialize unikernels to serverless workloads, based on the following observations: every serverless workload can be reduced to one or more ETL functions (see § 3), and that they are typically written in managed and/or interpreted languages (around 91% of serverless functions are written in python or node.js [20]). Extract, Transform and Load (ETL) [31] is a design pattern that succinctly captures event-driven execution, which is a typical use case for serverless functions: data is *extracted* from a source, a sequence of *transformations* are applied, and the result is stored (load). Specializing for ETL-style workloads, allows the unikernel to be more efficient than containers [39] while providing stronger isolation guarantees. ETL workloads have limited interaction with networking and storage, enabling optimizations that replace traditional but overly general network and file system APIs with high level APIs designed for ETL. We leverage the strong preference for managed and/or interpreted languages to reduce the time between invocation and execution by maintaining a pool of unikernels that are initialized with the required runtime. With these optimizations, a serverless platform based on unikernels and hypervisor virtualization can reduce overheads further, enabling much higher density and higher performance than current solutions while maintaining similar isolation guarantees.

2 Background

Serverless providers must guarantee three properties:

Isolation: A function or user must not be able to access data from other functions or users and must not be able to impact (maliciously or otherwise) the resources allocated to and used by others. AWS currently handles this with per-user VMs, each of which runs one or more functions for that user. Google's gVisor relies on the host kernel to handle memory management and scheduling for each function. VMs are arguably more secure than containers [1, 16]: the interface between the guest OS and the hardware (ABI) is narrower and easier to secure than the OS API. The number of vulnerabilities in the Linux kernel reported to the CVE database [9] (170 just in 2018) far outstrips the number of vulnerabilities found in hypervisors (4 for *KVM* in 2018).

Elasticity: Resource allocation should dynamically adapt to varying workload requirements. This is typically realized by automatically balancing load across providers' data centers. For example, on AWS Lambda, a user initially has no VMs. When a function is invoked, a pre-created idle VM is assigned to the user; subsequent function invocations are executed in that VM or cause another idle VM to be assigned to the user for execution. As functions finish, VMs are reclaimed and destroyed [47].

Startup Latency: The delay between when a function is invoked and when its execution starts (response time) should be as short as possible. The worst response time case occurs in the event of a cold start: the user has no execution environment setup and/or the function is not cached and needs to be initialized. Choice of programming language impacts cold start. For example, the warm and cold start time for a python 2.7 AWS Lambda function are 169 ms and over a second [47], respectively. The ExCamera [28] authors observed worst-case cold start time of over a minute when over 2000 functions were rapidly invoked.

3 Unikernels for Serverless

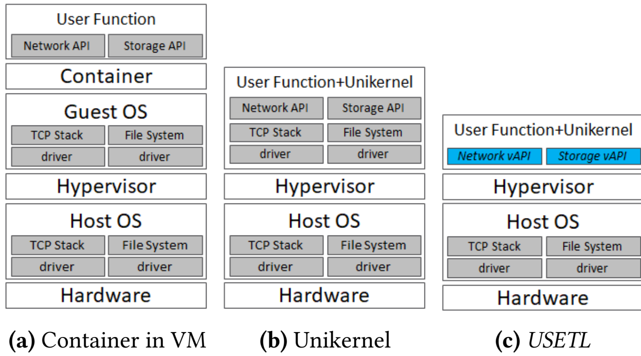
Unikernels comprise a minimal operating system and a single application, making them a natural fit for serverless functions [33]. By using unikernels, we eliminate redundant virtualization layers relative to containers within VMs, and avoid double scheduling and redundant memory consumption. Due to the unikernel's small footprint, the number of functions a server can execute concurrently (*function density*) is increased. Unikernels operate in a single address space, which implicitly avoids the costly overhead of mode switches [44].

Before a serverless function can be run, the client provides the service provider with function code and its dependencies. At this point, if the provider uses unikernels, they can bake the function code with the minimal OS and store the resulting unikernel. When the function is invoked, the service provider spins up a copy of the unikernel on a hypervisor.

Although a unikernel is relatively small, booting one requires unavoidable work because the virtual machine must be created. Virtual CPUs and network devices must be configured, memory must be reserved, and block devices must be mounted. While creating and booting a VM takes over a second (without any optimizations), this step can be moved off the critical path by creating the VMs beforehand, yielding a pool of unikernels ready to run. We analyze function startup time for both creating on-demand and using a pre-created environment in Section 4.

Pre-creating a pool of *all* registered (possibly millions of different) functions would not be feasible due to memory

[1] The mode switches can be reduced by, for example, putting device emulation in the kernel with vhost-net https://lwn.net/Articles/346267/



(a) Container in VM **(b)** Unikernel **(c)** *USETL*

Figure 1. Comparison of virtualization stack to provide serverless functions of current technique and unikernels.

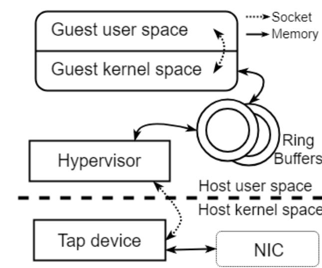


Figure 2. Guest networking through para-virtualized devices on the guest. The guest and hypervisor communicate through shared ring buffers, the hypervisor sends packets through a tap device to the kernel for handling.

and CPU constraints. This is a further challenge for elasticity — multiple instances of a function might need to run concurrently.

Over 96% of applications using serverless functions in 2018 were written in managed/interpreted languages like node.js, Python and Java [20]. We use this information to specialize our unikernels at the runtime level. The idea is to bake a minimal OS with the required runtime, and have it listen on a communication channel for the function to execute. When the host wants to execute a function, it loads the function at a specific guest address and signals the unikernel. For simplicity, we focus on Python in the rest of the paper, but our approach could be applied to any managed/interpreted language. We call this unikernel baked with an interpreter a *USF* (unikernel for serverless functions). When a function finishes, the *USF* that housed it does not need to be destroyed. Instead, the hypervisor can restore it to a checkpoint of its initial ready state, while making sure all residual data touched by the function is erased. This process resembles the *snapshot-rollback* mechanism in LwC [36], and is cheaper and far less complex than restoring a full VM. The hypervisor restores the initial image of the unikernel in a fixed memory range, while the rest is zeroed out. Moreover, as we discuss in greater detail later, the unikernel initially has no network state and storage is ephemeral, so they are simply discarded.

Specializing unikernels at the runtime layer and recycling are simple optimizations that make unikernels more suitable for serverless functions. In the following subsections, we propose more radical changes. First, we leverage the fact that the service provider controls the entire virtualization stack (Figure 1), from the hypervisor to the runtime, enabling them to optimize across layers. By construction, the user is completely unaware of where and how the function is being executed. We take advantage of this to place the Python interpreter closer to the hypervisor and redesign networking and storage by modifying the interface between the unikernel and the hypervisor.

3.1 Specializing for ETL

Serverless functions typically follow a common pattern: Input data to the serverless function comes from parameters or remote storage (e.g. retrieved through `HTTP GET` requests). After performing arbitrary computation on this input data, the function sends results back to the caller or stores it in remote storage (through `HTTP POST` requests). This is a subset of the **E**xtract, **T**ransform and **L**oad (ETL) [31] design pattern: data is *extracted* from a source, a sequence of *transformations* are applied, and the result is stored (load).

ETL-style workloads make up the majority of use case of serverless functions (at the time writing, all six use case examples on AWS Lambda's homepage are ETL). Further, all serverless workloads can be transformed to ETL due to the generality of the pattern. *USETL* does not limit functions to follow the ETL sequence strictly; functions can have any order of actions. *USETL*'s I/O optimizations are guided by characteristics of ETL workloads.

3.2 Network Devices

Network virtualization in most hypervisors — like KVM — is complex. Consider a guest application trying to send data to the outside world. The usermode application in the guest originates the request, which is then processed by the network stack in the guest kernel. The guest kernel network stack drives an emulated or para-virtualized network device. This usually results in a *vmexit*. The host kernel traps this request, and passes the request to the hypervisor, which then pipes the data into a tap device [2] so it can finally be sent out by the host kernel. Figure 2 shows a common way of setting up guest networking, where the guest and hypervisor communicate through shared memory. This is how a guest using *virtio* para-virtualized devices provides networking. Figure 3 shows how I/O requests reach the hypervisor from the guest in KVM, including the mentioned mode switches.

[2]A tap device is a virtual network interface (NIC) in which one end is attached to the kernel, similar to a regular physical NIC, while the other end is attached to a user level application. All traffic is still managed by a single kernel.

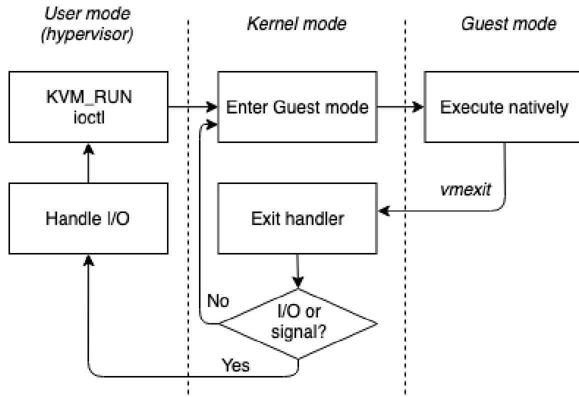


Figure 3. KVM introduces a new mode called guest. When the guest performs I/O, the kernel traps its execution and handles the request in the hypervisor. If it's a network request it could possibly be handed back to the kernel through a tap device. Based on a figure of the KVM documentation [12].

Unikernels implicitly remove the guest user to guest kernel mode switch [35] as they operate in guest kernel mode at all times. The guest kernel to host kernel mode switch can't be eliminated completely but can be made infrequent by batching, i.e., by only notifying (*kicking*, in *virtio* terms) the host kernel after a certain number of packets have been enqueued or after a certain time has passed. The host kernel to host user space mode switch can be eliminated by moving the device emulation into the host's kernel [23]. Both the host kernel to host user mode switch and the later host user to host kernel switch can be bypassed by using user level networking like DPDK [2] or SR-IOV passthrough [8]. Reducing mode switches is important in high performance applications and in high-density multi-tenant computing platforms (our case) because these overheads can add up and impact total effective CPU utilization [44]. Receiving network packets in the guest is simpler: when the host kernel gets a packet addressed to a guest, it is written to a tap device, received by the hypervisor on the other end of it and written to the guest's virtual network device. One of these two mode switches can be eliminated by moving the network entirely to either the kernel or to the user level hypervisor. These optimizations help but do not remove fundamental inefficiencies.

3.2.1 A different approach

Instead of optimizing the network execution flow, we propose something more radical: remove network devices entirely from *USFs* and instead use a higher level API between the unikernel and the host. By making the host handle networking on behalf of the *USF*, one entire network stack is skipped, *vmexits* are no longer proportional to the number of packets sent by the *USF*, and tap devices (a significant source of mode switches) are no longer necessary.

This idea works because of two characteristics of serverless workloads. First, serverless functions are not addressable: the function has to initiate all connections. Second, ETL-style functions do not need general networking capabilities. All that is necessary are operations similar to *HTTP* `GET` and `POST` (see § 3.1): `GET` takes the *url* of the resource/file to be fetched, while `POST` takes the destination *url* and a file. On a `GET` request, the host fetches the data on the *USF*'s behalf, shares the content through memory and notifies the *USF* of the data's location. On a `POST` request, the host sends data from the specified (memory) location of the file to be sent out, to the provided *url*. In both cases the files' contents are in memory; as we will discuss in the next subsection, *USETL* only uses an in-memory file system.

On the host side, we have two choices to handle network API requests: interrupts (*vmexits*) or polling [3, 34]. Although the best approach is ultimately workload dependent, we use interrupts because polling typically wastes CPU cycles. We expect that the number of interrupts will be small because the API operates at file granularity, instead of packet granularity like in a traditional network stack. Host CPU time spent handling API on behalf of function can be billed to the tenant.

Applications which use a centralized entity to manage and assign batch work (e.g. ExCamera [28] and Sprocket [25]) can easily be modified to fit this model. Instead of a function connecting to a monitor to retrieve input data, the monitor can invoke the function with a *url* to the data as a parameter. When the function begins executing, it loads data from that *url* through the `GET` API. If function is synchronous, the API gateway that handles invocation requests can act as an intermediary, receiving the result from the function and forwarding the result to the invoker when the function finishes.

Replacing networking with this API breaks backwards compatibility with existing general Python source code. However, the goal of serverless functions is not to execute arbitrary applications out of the box, but to provide elasticity and other features in exchange for adaptations to the FaaS model. Further, there is a strong incentive to use the new API: functions that finish sooner cost less. The service provider could also offer unikernels with an unmodified runtime to provide backwards compatibility with existing serverless functions.

3.3 Storage and File System

Serverless functions are stateless by design: an event triggers the creation of a function, with optional parameters, on an arbitrary computing node. The function executes a short task and terminates. The orchestration system destroys or recycles the execution context, and reclaims all resources allocated to the function. When state is unavoidable it must be stored and retrieved from remote storage. Functions are typically short-lived, and have ephemeral and limited local storage: AWS Lambda limits function execution to 15

minutes and provisions each function with only 512 MB of temporary storage.

Consequently, the storage subsystem in a *USF* can be greatly simplified: most of the guarantees and features (e.g., journaling, crash recovery, sharing, access control, etc.) provided by a full-fledged file system like ext4 [41] are not only unnecessary, but also contradictory.

With large and cheap non-volatile memory (NVM) devices with memory-like characteristics on the horizon, we hypothesize that service providers will stop providing 'persistent' storage to serverless functions. Functions will operate only on memory. AWS Lambda providing environments with a small fixed amount of ephemeral storage but with up to 3 GB of memory indicates that memory-only storage is already here.

Simple object storage can instead be provided through the network API. GET and POST can be used to read and write *urls* backed by host files. Where necessary, more complex storage can be provided by an ephemeral in-memory file system with a POSIX interface, to support legacy applications and libraries. The shared memory used by the file system is split into two regions, each with its own offset table: one for files created by the *USF* and one for files created by the host. Separating the file system into two areas ensures that each only has a single producer, removing the necessity for complex mutual exclusion mechanisms. Further, this design maps cleanly into the network API: when the *USF* requests a non-local file, the host downloads the content of the *url* directly into the host region of the file system and then maps the file readable to the *USF*. When the *USF* wishes to perform a write, it writes the data into it's region of the shared in-memory file system, and provides the filename to the POST request. The host locates the file offset and then sends the data directly to the destination *url*. If a *USF* tries to modify a file in the host's region, a copy-on-write mechanism is triggered, copying the file to the *USF*'s region.

By removing the traditional storage stack from the unikernel image, we reduce the *USF*'s memory footprint and the amount of CPU time spent per I/O operation.

3.4 General Optimizations

With networking and storage redesigned, we focus now on issues on the host and related to Python. One of them is Python module management: different applications may require different Python modules/versions and we want to avoid having to install them when a function is being invoked. Another problem is that if all *USFs* allocate and have their own data without sharing, memory will be wasted due to duplication.

We solve module dependency in a simple way: each serverless host keeps the default Python modules (22 MB for Python 3.7) and the most commonly used external Python modules in memory (highly skewed, 36% of imports are for just 20 packages, 0.02% of the PyPI repository [42]). Since the each

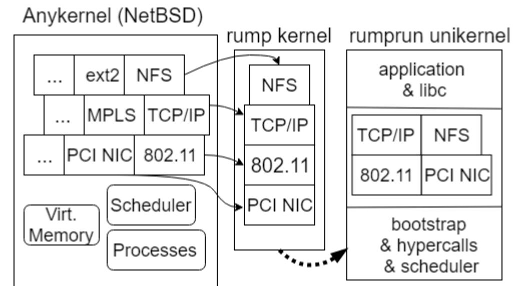


Figure 4. Rump kernels make it possible to pick and choose components of NetBSD (an anykernel), form kernels with a particular interface and attach an application to produce a rumprun unikernel [17] that can run on bare-metal or on a hypervisor.

function's dependencies are explicitly indicated when a function is deployed, each host can maintain a per-user module directory, which contains links to a central module directory to avoid duplication. When a function is assigned to a *USF*, the host shares the dependencies with that *USF* through the in-memory file system.

Because all unikernels are running the same application - the Python interpreter - we can deduplicate non-private regions of memory to reduce usage, consequently enabling a higher function density. An existing mechanism like kernel same-page merging (KSM) [3] , which scans regions of memory marked as deduplication candidates and merges/ shares them to save memory usage could be used, but there are problems: the kernel has to keep scanning pages to find similar ones, potentially wasting CPU cycles. We avoid scanning pages by noting that the hypervisor has complete knowledge of sections that are duplicated when it is creating the *USFs*. The final goal is to make all running unikernels with a particular runtime share the same text and code sections, which would greatly reduce TLB overhead, while each has its own data sections.

4 Preliminary Evaluation

For preliminary evaluation, we used a modified rumprun unikernel [19], which is based on rump kernels [18, 29] (Figure 4). The rumprun unikernel is customizable and has a large repository of common applications, among them, Python. Other available unikernels are Mirage [38], OSv [15], ClickOS [40] and UKL [22], each with its own different goals and languages supported.

As a preliminary method to measure the improvement from changing the unikernel's network stack to an API, we measured the time taken to download 2, 8, 64 and 256 MB files in the following settings: in a unikernel with hard drive backed storage, and on the host to both hard drive and

[3]Kernel same-page merging: https://www.kernel.org/doc/html/latest/admin-guide/mm/ksm.html

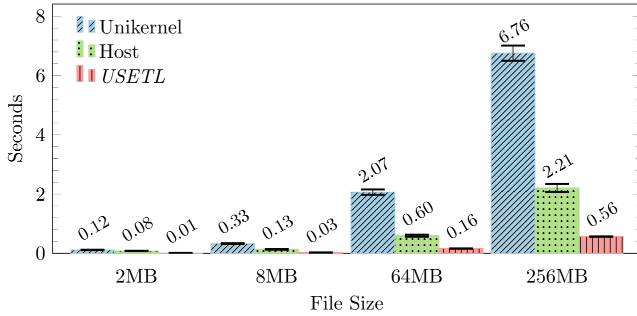


Figure 5. Elapsed time in seconds to download a file in the unikernel versus downloading on the host.

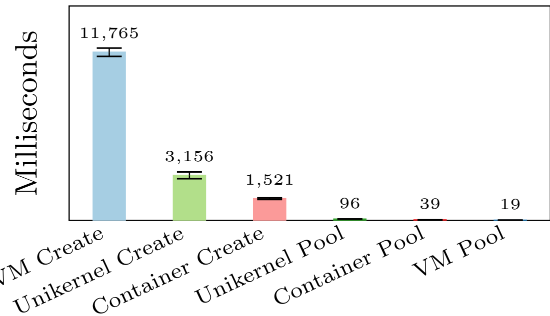


Figure 6. Cold start time for a simple connect and quit function under different environments. *Create* refers to when the environment was created on demand, while pool refers to the setup when the environment was pre-created and had a runtime listening for requests. *USETL* uses unikernel pools.

memory-backed storage. The memory backed storage case approximates *USETL*. To avoid external network interference, we set up a simple HTTP server on the same machine, which served all experiments without change, and repeated every test 30 times. Figure 5 shows the mean and standard deviation of elapsed time. The data shows that the extra layer of virtualization and network stack affects the elapsed time substantially, and that moving networking from the unikernel to the host can be beneficial; also, as expected, mapping the incoming file directly to a filesystem in memory is much faster than using a local disk.

In order to substantiate our claim regarding the need to remove the creation of the environment from the critical path, we also measured cold start time in the following environments: a plain Ubuntu 16.04 VM, a container and a unikernel. We measured actual cold start time by creating the the environment when the function was invoked; and warm-start time (labeled pooled), where the environment was pre-created and was listening on a socket for the serverless function to execute. The function in question is trivial: it merely connects to the host and then terminates. We measured the time elapsed between when the host launched the function and when it received the connection from the function, averaged over 30 runs. Figure 6 shows that while it takes longer to create a VM than to create a unikernel due to size, creating a unikernel is slower than creating a container (the container is just a process after all). However, in all cases, environment creation takes too long to be feasible on the critical function invocation's path. For pooled environments, all three have environments showed similar times since the runtime is pre-created and just listens on a socket. Minor variations observed were likely due to network set up.

To analyze memory usage, we called the *smem* linux tool from a python script that simply loops forever. A Docker container process uses 28.7 MB and the unikernel plus hypervisor uses 54.1 MB of unique (non-shared) memory; the unikernel reports a total usage of 10.3 MB during boot, excluding virtio buffers. We believe that most of this 10.3 MB used by a unikernel can be shared by many *USFs*, leaving the rest of the memory to per-function allocation.

Although disk space is not a concern, we report it for completeness: the unikernel with python *elf* file has a size of 11 MB (aligning with the memory usage reported during boot). Stripping symbols from the python interpreter brought this down to 6.5 MB. The kernel consumes 4.5 MB. The script itself is 112 B, but it's passed to the unikernel as a 352 kB iso image. The container's size consists of just the script itself (112 B) plus 922 MB shareable among different python containers.

5 Conclusion

Unikernels provide the same isolation guarantees as running a container within a VM while using less resources, and can be optimized for serverless functions by replacing the general I/O interface with a ETL-specific API interface. Since all unikernels that run a language interpreter like python are interchangeable, they can be pre-created and pooled. Further, common memory regions can be deduplicated, and only function specific memory needs to be private. With these modifications, per-function CPU and memory usage is reduced on the host, thereby increasing the number of functions that can be executed concurrently, translating to lower operating costs for the service provider.

6 Acknowledgments

We thank the anonymous reviewers and SCEA group members for their feedback. This research was partially supported by the NSF (CNS-1618563).

References

[1] [n. d.]. 1631: gVisor runsc guest->host breakout via filesystem cache desync. https://bugs.chromium.org/p/project-zero/issues/detail?id=1631. Accessed: 2019-03-25.

[2] [n. d.]. 43. Vhost Library - Dataplane Development Kit. https://doc.dpdk.org/guides-18.08/prog_guide/vhost_lib.html Accessed: 2019-03-01.

[3] [n. d.]. abelg/virtual_io_acceleration: Virtual I/O acceleration technologies for KVM. https://github.com/abelg/virtual_io_acceleration. Accessed: 2019-03-10.

[4] [n. d.]. AWS Lambda - Serverless Compute. https://aws.amazon.com/lambda/. Accessed: 2019-03-10.

[5] [n. d.]. Azure Functions - Serverless Architecture. https://azure.microsoft.com/en-us/services/functions/. Accessed: 2019-03-10.

[6] [n. d.]. Cloud Functions - Event-driven Serverless Computing. https://cloud.google.com/functions/. Accessed: 2019-03-10.

[7] [n. d.]. Cloud Functions - Overview. https://www.ibm.com/cloud/functions. Accessed: 2019-03-10.

[8] [n. d.]. Configure SR-IOV Network Virtual Functions in Linux KVM. https://software.intel.com/en-us/articles/configure-sr-iov-network-virtual-functions-in-linux-kvm. Accessed: 2019-04-24.

[9] [n. d.]. CVE security vulnerability database. https://www.cvedetails.com/product/47/Linux-Linux-Kernel.html?vendor_id=33. Accessed: 2019-03-05.

[10] [n. d.]. Firecracker - Lightweight Virtualization for Serverless Computing. https://aws.amazon.com/blogs/aws/firecracker-lightweight-virtualization-for-serverless-computing/. Accessed: 2019-01-28.

[11] [n. d.]. google/gvisor: Container Runtime Sandbox. https://github.com/google/gvisor. Accessed: 2019-03-25.

[12] [n. d.]. kvm: the Linux Virtual Machine Monitor. https://www.kernel.org/doc/ols/2007/ols2007v1-pages-225-230.pdf. Accessed: 2019-05-03.

[13] [n. d.]. Measuring the Horizontal Attack Profile of Nabla Containers. https://blog.hansenpartnership.com/measuring-the-horizontal-attack-profile-of-nabla-containers/. Accessed: 2019-03-25.

[14] [n. d.]. Networking — KVM. https://www.linux-kvm.org/page/Networking. Accessed: 2019-07-15.

[15] [n. d.]. OSv - the operating system designed for the cloud. http://osv.io/. Accessed: 2019-02-27.

[16] [n. d.]. Privilege Escalation in gVisor, Google's Container Sandbox. https://justi.cz/security/2018/11/14/gvisor-lpe.html. Accessed: 2019-03-25.

[17] [n. d.]. Repo rumpkernel/wiki Wiki. https://github.com/rumpkernel/wiki/wiki/Repo. Accessed: 2019-03-05.

[18] [n. d.]. Rump Kernels. http://rumpkernel.org/. Accessed: 2019-03-05.

[19] [n. d.]. rumpkernel/rumprun: The Rumprun unikernel and toolchain for various platforms. https://github.com/rumpkernel/rumprun. Accessed: 2019-03-05.

[20] [n. d.]. Serverless by the number: 2018 report. https://serverless.com/blog/serverless-by-the-numbers-2018-data-report/. Accessed: 2019-02-27.

[21] [n. d.]. State of the Cloud Report. https://www.rightscale.com/lp/state-of-the-cloud. Accessed: 2019-01-28.

[22] [n. d.]. UKL: A Unikernel Based on Linux - now + next. https://next.redhat.com/2018/11/14/ukl-a-unikernel-based-on-linux/. Accessed: 2019-02-27.

[23] [n. d.]. vhost_net: a kernel-level virtio server. https://lwn.net/Articles/346267/. Accessed: 2019-04-24.

[24] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. 2018. SAND: Towards High-Performance Serverless Computing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 923–935. https://www.usenix.org/conference/atc18/presentation/akkus

[25] Lixiang Ao, Liz Izhikevich, Geoffrey M. Voelker, and George Porter. 2018. Sprocket: A Serverless Video Processing Framework. In *Proceedings of the ACM Symposium on Cloud Computing, SoCC 2018, Carlsbad, CA, USA, October 11-13, 2018*. 263–274. https://doi.org/10.1145/3267809.3267815

[26] Sean Barker, Timothy Wood, Prashant Shenoy, and Ramesh Sitaraman. 2012. An Empirical Study of Memory Sharing in Virtual Machines. In *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*. USENIX, Boston, MA, 273–284. https://www.usenix.org/conference/atc12/technical-sessions/presentation/barker

[27] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. 1995. Exokernel: An Operating System Architecture for Application-level Resource Management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles (SOSP '95)*. ACM, New York, NY, USA, 251–266. https://doi.org/10.1145/224056.224076

[28] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. 2017. 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)*. USENIX Association, Boston, MA, 363–376. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/fouladi

[29] Antti Kantee and Justin Cormack. 2014. Rump Kernels: No OS? No Problem! *;login:* 39, 5 (2014). https://www.usenix.org/publications/login/october-2014-vol-39-no-5/rump-kernels-no-os-no-problem

[30] Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. 2018. Scaling Guest OS Critical Sections with eCS. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 159–172. https://www.usenix.org/conference/atc18/presentation/kashyap

[31] Ralph Kimball, Laura Reeves, Warren Thornthwaite, Margy Ross, and Warren Thornwaite. 1998. *The Data Warehouse Lifecycle Toolkit: Expert Methods for Designing, Developing and Deploying Data Warehouses with CD Rom* (1st ed.). John Wiley & Sons, Inc., New York, NY, USA.

[32] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. 2018. Pocket: Elastic Ephemeral Storage for Serverless Analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 427–444. https://www.usenix.org/conference/osdi18/presentation/klimovic

[33] Ricardo Koller and Dan Williams. 2017. Will Serverless End the Dominance of Linux in the Cloud?. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems (HotOS '17)*. ACM, New York, NY, USA, 169–173. https://doi.org/10.1145/3102980.3103008

[34] Yossi Kuperman, Eyal Moscovici, Joel Nider, Razya Ladelsky, Abel Gordon, and Dan Tsafrir. 2016. Paravirtual Remote I/O. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*. ACM, New York, NY, USA, 49–65. https://doi.org/10.1145/2872362.2872378

[35] Wenhao Li, Yubin Xia, Haibo Chen, Binyu Zang, and Haibing Guan. 2015. Reducing World Switches in Virtualized Environment with Flexible Cross-world Calls. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture (ISCA '15)*. ACM, New York, NY, USA, 375–387. https://doi.org/10.1145/2749469.2750406

[36] James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, Bobby Bhattacharjee, and Peter Druschel. 2016. Light-Weight Contexts: An OS Abstraction for Safety and Performance. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 49–64. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/litton

[37] Anil Madhavapeddy, Thomas Leonard, Magnus Skjegstad, Thomas Gazagnaire, David Sheets, Dave Scott, Richard Mortier, Amir Chaudhry, Balraj Singh, Jon Ludlam, Jon Crowcroft, and Ian Leslie. 2015. Jitsu: Just-In-Time Summoning of Unikernels. In *12th USENIX*

Symposium on Networked Systems Design and Implementation (NSDI 15). USENIX Association, Oakland, CA, 559–573. https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/madhavapeddy

[38] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. 2013. Unikernels: Library Operating Systems for the Cloud. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*. ACM, New York, NY, USA, 461–472. https://doi.org/10.1145/2451116.2451167

[39] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. 2017. My VM is Lighter (and Safer) Than Your Container. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. ACM, New York, NY, USA, 218–233. https://doi.org/10.1145/3132747.3132763

[40] Joao Martins, Mohamed Ahmed, Costin Raiciu, and Felipe Huici. 2013. Enabling Fast, Dynamic Network Processing with clickOS. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN '13)*. ACM, New York, NY, USA, 67–72. https://doi.org/10.1145/2491185.2491195

[41] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, and Laurent Vivier. 2007. The new ext4 filesystem: current status and future plans. In *Proceedings of the Linux symposium*, Vol. 2. 21–33.

[42] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. SOCK: Rapid Task Provisioning with Serverless-Optimized Containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 57–70. https://www.usenix.org/conference/atc18/presentation/oakes

[43] Diego Ongaro, Alan L. Cox, and Scott Rixner. 2008. Scheduling I/O in Virtual Machine Monitors. In *Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '08)*. ACM, New York, NY, USA, 1–10. https://doi.org/10.1145/1346256.1346258

[44] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. 2014. Arrakis: The Operating System is the Control Plane. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO, 1–16. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/peter

[45] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. 2019. Shuffling, Fast and Slow: Scalable Analytics on Serverless Infrastructure. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 193–206. https://www.usenix.org/conference/nsdi19/presentation/pu

[46] Xiang Song, Jicheng Shi, Haibo Chen, and Binyu Zang. 2013. Schedule Processes, Not VCPUs. In *Proceedings of the 4th Asia-Pacific Workshop on Systems (APSys '13)*. ACM, New York, NY, USA, Article 1, 7 pages. https://doi.org/10.1145/2500727.2500736

[47] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. 2018. Peeking Behind the Curtains of Serverless Platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 133–146. https://www.usenix.org/conference/atc18/presentation/wang-liang

[48] Chuliang Weng, Zhigang Wang, Minglu Li, and Xinda Lu. 2009. The Hybrid Scheduling Framework for Virtual Machine Systems. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '09)*. ACM, New York, NY, USA, 111–120. https://doi.org/10.1145/1508293.1508309