SEUSS: Skip Redundant Paths to Make Serverless Fast

James Cadden
jmcadden@bu.edu
Boston University

Thomas Unger
tommyu@bu.edu
Boston University

Yara Awad
awadyn@bu.edu
Boston University

Han Dong
handong@bu.edu
Boston University

Orran Krieger
okrieg@bu.edu
Boston University

Jonathan Appavoo
jappavoo@bu.edu
Boston University

ABSTRACT

This paper presents a system-level method for achieving the rapid deployment and high-density caching of serverless functions in a FaaS environment. For reduced start times, functions are deployed from unikernel snapshots, bypassing expensive initialization steps. To reduce the memory footprint of snapshots we apply page-level sharing across the entire software stack that is required to run a function. We demonstrate the effects of our techniques by replacing Linux on the compute node of a FaaS platform architecture. With our prototype OS, the deployment time of a function drops from 100s of milliseconds to under 10 ms. Platform throughput improves by 51x on workload composed entirely of new functions. We are able to cache over 50, 000 function instances in memory as opposed to 3, 000 using standard OS techniques. In combination, these improvements give the FaaS platform a new ability to handle large-scale *bursts* of requests.

ACM Reference Format:
James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. 2020. SEUSS: Skip Redundant Paths to Make Serverless Fast. In *Fifteenth European Conference on Computer Systems (EuroSys '20), April 27–30, 2020, Heraklion, Greece.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3342195.3392698

1 INTRODUCTION

Computational caching is a strategy systems can employ to address computational redundancy by applying pre-computed state to advance running execution [43]. At the heart of any computational caching approach is a change in the system's representation of execution that converts computational state into a data object amenable to exposing redundancies [7, 13, 14, 27].

One application model that is particularly amenable to computational caching is that of *serverless functions*, wherein high-level scripts are deployed on-demand by a remote FaaS platform. For example, the function start time can be shortened by using a language interpreter that has already been initialized with the source code and dependencies of the function [20].

The traditional caching techniques employed by FaaS platforms (i.e., holding idle interpreter processes isolated within a dedicated container or VM) are resource-intensive [25]. Consequently, research systems have demonstrated lightweight methods for computational caching within FaaS platforms, such as deploying execution from images of checkpointed processes [39] or forking execution from existing processes [15, 19]. Fundamentally, these systems demonstrate that caching can both shorten execution start times and reduce the memory footprints of cached function instances. In our opinion, any user-level caching approach will be insufficient as a general solution for serverless platforms. For example, forking requires cooperation from within the interpreter, which limits the set of interpreters that can be used. In addition, kernel-managed state required for execution sandboxing cannot be captured at user-level, therefore, extending the path of a deployment.

In this paper, we present *Serverless Execution via Unikernel SnapShots* (SEUSS), a method for achieving the rapid deployment and high-density caching of serverless functions in a multi-tenant FaaS environment. The goal of SEUSS is to provide a caching solution for serverless functions that supports a diverse set of language runtimes, executes functions in isolation, and efficiently captures computational state across both the application and system levels. We achieve these properties by deploying functions from *unikernel snapshots*.

In SEUSS, function logic is packed with a language interpreter and library OS into an isolated unikernel. The flat address space of the unikernel enables a straightforward

EuroSys '20, April 27–30, 2020, Heraklion, Greece

ACM ISBN 978-1-4503-6882-7/20/04…$15.00
https://doi.org/10.1145/3342195.3392698

method for capturing and caching the entire memory footprint and register state of the function. Function state can be captured in a black-box fashion at an arbitrary point during execution, as an in-memory *snapshot* image. New execution can be rapidly deployed from a snapshot, dramatically shortening the function's start time by skipping the following paths: booting the unikernel, initializing the language runtime, and importing and compiling the function code and dependencies.

Dedicating an entire system stack for each individual function introduces large amounts of replicated state. SEUSS is able to significantly reduce the memory use of functions by applying page-level sharing ubiquitously across the entire software stack that is required to run high-level function code (i.e., the application logic, language interpreter, user libraries, and supporting kernel). This reduction not only fully recovers the cost of replicating kernel state per function, but greatly improves over the current techniques of platform function caching.

Through our method, anticipatory optimizations are achieved by preemptively warming the internal pathways and data structures of the unikernel stack prior to capturing a snapshot image. This intuitive technique has the dual benefit of reducing the memory footprint and shortening the start time of execution deployed from a snapshot (§3).

We implement SEUSS within a prototype kernel and evaluate its benefit within the context of a full-feature FaaS platform architecture. Replacing Linux with our SEUSS kernel, we are able to improve the number of cached function contexts from 3,000 to over 54,000 on a single compute node. The cost of a cache miss for a function drops from 500 ms to 7 ms. Using SEUSS, we demonstrate support for bursts of requests with little impact on background workloads, whereas the Linux implementation fails to adequately support either.

Our key insight is that changing the representation of execution enables powerful optimizations to squash redundancy across both application and systems pathways. We assert that the SEUSS method is simple and general enough that it can be readily adopted by production-grade systems.

2 BACKGROUND

In the Function-as-a-Service (FaaS) model, a *serverless function* is a short code segment written for a high-level language interpreter, such as Node.js, Python, or Java. In this model, functions are executed by a remote FaaS platform in response to invocation requests. Serverless functions promote application designs that are composed out of many short-lived executions, which are deployed rapidly as singletons, in sequences, or in parallel [20]. For programmers, serverless functions offer a powerful primitive for accessing on-demand computation across arbitrary scales [21].

For the FaaS platform, the operational shift away from the application developer allows the system to control how functions are deployed. For example, functions can be deployed within dedicated containers [17], virtual machines [6, 33, 40], process-level encapsulation [49], or language-level isolation [10]. To simplify management, client functions are typically restricted to a small set of language interpreters.

Booting interpreters and compiling source code can lead to long initialization overheads for function deployments. Therefore, to achieve low-latency start times, the FaaS platform must cache intermediate stages along a function's invocation lifetime for reuse (Figure 1). To enable fast deployments for many different functions, FaaS platforms are required to hold large amounts of state cached in memory. For example, by assuming a small set of interpreters, platforms can manage pre-initialized pools of running interpreters (T1 in Figure 1). Moreover, the stateless nature of functions enables the platform to cache the execution environment of a specific function (e.g., the container or VM where the function ran) for immediate reuse across future invocations (T2 in Figure 1). The requirement to cache many isolated environments in memory, many of which are limited in reuse to single function, presents a unique challenge that modern operating systems have been slow to address [25].

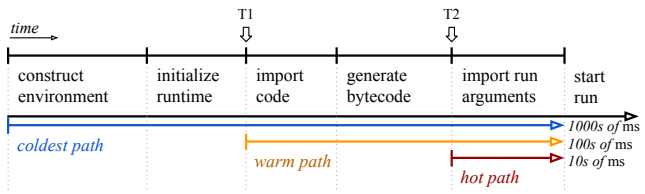

Figure 1: Stages of a function invocation

3 SEUSS METHOD

SEUSS is a system-level method for the rapid deployment of serverless functions in a multi-tenant FaaS environment. Its goal is to provide a general solution for serverless functions that supports a diverse set of language runtimes, executes functions in isolation, and efficiently caches application and system state to enable fast deployments. We achieve these desired properties by deploying functions from *unikernel snapshots*.

Unikernel Contexts (UCs)

Our key insight is that the act of *unikernelization* transforms process-like computation into a format that is amenable to computational caching. Unikernels provide the mechanism that we use to encapsulate and observe the execution state of a serverless function wherever it manifests in the software stack.

In a unikernel, the application and its supporting system functionality (e.g., file system, user libraries, and network stack) are combined into a single, flat address space. In SEUSS, each *unikernel context* (UC) consists of a high-level language interpreter (e.g., Node.js, Python) configured to import and execute function code. UCs act as our unit of deployment for individually isolated function executions, the security implications of which are discussed in §5.

The apparent drawback of unikernelization is that it can introduces large amounts of replicated state and redundant initialization procedures by replicating state and functionality that was once shared. However, these redundancies are detected and reduced through our use of *snapshots*, *snapshot stacks*, and *anticipatory optimizations*.

Snapshots

A snapshot is an immutable data object which expresses the instantaneous execution state of a UC (i.e., its address space and registers). In SEUSS, snapshots act as templates from which UCs can be deployed. Since snapshots are immutable memory images, an arbitrary number of UCs can be launched from a single snapshot concurrently and over time. By capturing a fully-initialized UC in a base snapshot, every deployed UC can avoid the overheads of booting the unikernel and setting up an interpreter.

Encapsulating execution in a UC acts to minimize the amount of execution state external to the UC. Hence, the actions of snapshotting and deploying from a snapshot become simple operations on address spaces via their backing data structures. Furthermore, when snapshots can be triggered and gathered externally, the act of capturing a UC requires no semantic understanding of the internal state of the UC. In other words, snapshots can treat the software within the UC as a black box. Our approach stands in contrast to a fork-based approach which requires explicit application-level support and coordination with the kernel. For example, a snapshot approach works identically across different interpreters, including interpreters that do not natively support fork (such as Node.js).

Snapshot Stacks

Snapshot stacks express a lineage between snapshots across time, much like a fork tree captures a lineage relationship of processes. Snapshot stacks work by treating each snapshot as a page-level *diff* on the previous snapshot in a snapshot stack. For this, we use traditional copy-on-write semantics to capture into a snapshot only the pages that were modified by the target UC.

Snapshot stacks are designed to increase the number of functions that can be cached in memory by factoring out common execution state shared across the UCs. To better understand the advantage of snapshot stacks, consider the following example: a FaaS platform wants to snapshot the fully-initialized state of JavaScript functions `Foo()` and `Bar()`. Armed with only the snapshot mechanism, the platform requires two UC snapshots, one for each function. If the interpreter is 100MB and each function adds 1MB, we require 202 MB of storage. With snapshot stacks, three snapshots are used, one for the initialized JavaScript interpreter, a second for the `Foo()` diff and a third for the `Bar()` diff. This requires 102 MB as the interpreter is shared between the two function snapshots.

Anticipatory Optimizations

We define Anticipatory Optimization (AO) as the act of intentionally running computation prior to capturing a snapshot with the goal of removing redundant space and time usage from subsequent execution. Functions based on the same managed runtime will exercise similar pathways to setup and execute a function. AO provides the opportunity to further migrate initialization procedures and memory into the shared snapshots. The redundancies we target include dynamically-generated data structures internal to the interpreter or subsystems of the unikernel, which would otherwise require significant effort to optimize-out manually.

In SEUSS, we apply AO by preemptively warming the internal pathways and data structures of the UC software stack prior to capturing the base runtime snapshot. AO decreases both the start up and execution times by enabling executions spawned from snapshots to avoid costly allocation and initialization procedures. Furthermore, the technique improves the cacheability of function-specific snapshots by reducing the number of written pages captured in each snapshot. With AO, the memory footprint of the base snapshot grows, but this space trade-off results in the improved performance and decreased footprint of all snapshots and UC instances descended from the base snapshot. See §7 for our evaluation of AO.

Our experience with applying AO to our unikernel environments suggests it provides an intuitive tool for enacting significant performance optimizations. We use semantic knowledge about the task of deploying high-level function code to anticipate and pre-execute common procedures. For example, we explored the benefit of sending TCP traffic into the unikernel, and running a (dummy) script through the JavaScript interpreter prior to capturing the base snapshot. These two AOs resulted in multiplicative reductions in warm and cold function execution times (illustrated in Figure 1), as well as the doubling of the snapshot cache density (§7). Importantly, the AOs we employ were discovered through basic reasoning about the high-level procedure of importing

and deploying function code, without knowledge or concern for the subtle inner workings of the interpreter and kernel.

4 SEUSS IN ACTION

This section describes the high-level procedure for deploying serverless functions using SEUSS on a FaaS compute node, as illustrated in Figure 2.

As part of the early system initialization, the unikernel is booted into the language interpreter and an *invocation driver* (script) is run, which creates HTTP/REST endpoint. The system sends commands and function code into the unikernel via this script. A runtime snapshot is taken (Ⓑ in Figure 2) after the invocation driver has started. These runtime snapshots may be relatively large in memory use (hundreds of MBs) but there are few of them: only one per supported interpreter. When a new UC is deployed from a snapshot, the internal driver is started in a listening state ready to accept a new connection.

To deploy serverless functions, SEUSS maintains a cache of snapshots as well as a cache of idle UCs. Function invocation requests are received from a remote FaaS platform controller, after which the system can take one of three paths to process each invocation: cold, warm, or hot. When no cached snapshot exists for the particular function being invoked, its UC is deployed from the base runtime snapshot, and the function source is imported into the UC and interpreted by the runtime (Ⓒ in Figure 2). Once the function source compilation step is completed, a function-specific snapshot is captured (Ⓢ in Figure 2). Next, the run arguments are imported into the UC, and the execution of the function begins, completing a cold path invocation.

When a function-specific snapshot exists for an invoked function, the warm path is taken by creating a UC from that snapshot, skipping the code import and compilation stages, importing the run arguments, and finally beginning function execution (Ⓦ in Figure 2). Once the execution of a function has finished, its UC can either be destroyed or cached for future invocations of that function on a new set of arguments. The hot invocation path consists of importing a new set of run arguments into an already constructed UC (Ⓗ in Figure 2).

5 SECURITY

Security must be a central design concern in multi-tenant cloud environments. In SEUSS, unikernels are used to isolate co-running function instances from each other and from the trusted kernel. Our prototype uses standard x86 user/kernel protection domains to isolated the untrusted UCs (ring 3) from the trusted OS (ring 0). Our approach is compatible with other hardware-enforced protection mechanisms, such as isolating unikernels within virtual machines (§9).

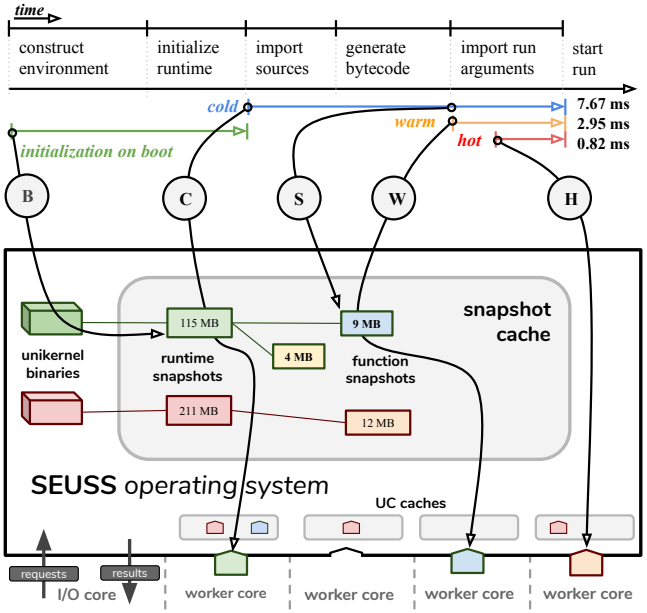

Figure 2: The high-level operations of a SEUSS OS deployed on a FaaS compute node.

Additional protection is achieved by narrowing the domain interface between the untrusted unikernel and the host to a restricted set of system calls [18, 36, 47, 49]. A narrow domain interface limits the surface area that a compromised guest can use to launch attacks (i.e., the majority of kernel functionality is handled *within* the isolated domain) and allows the system to more easily monitor and detect malicious behavior. For example, the hypercall interface used in our prototype, `ukvm` [47], exposes only 12 system calls while the standard security of a Docker container gives access to over 300 Linux syscalls [3].

Our use of snapshots preserves isolation between guests by restricting sharing to read-only pages within the historical timeline of a function. Runtime snapshots are captured before any function-specific information has been imported into the unikernel, allowing functions of different users to share the same base snapshot. UCs deployed from these snapshots have copy-on-write access to the shared set of read-only pages within the snapshot. Therefore, all writes are captured onto new pages that are dedicated exclusively to the UC that issues them.

The pervasive use of page-sharing does mean that our approach is susceptible to side-channel attacks and hardware-level vulnerabilities. For example, the Rowhammer exploit could be used to corrupt memory in a snapshot which is shared across many thousands of functions. Our approach is compatible with existing Rowhammer defenses [16]. In contrast to KSM [8], page-sharing in SEUSS is not applied

retroactively, reducing the concern for deduplication-based side-channel attacks [28]. However, the time to process a memory read can be used as a side-channel to leak information about the activity of co-running guests. While this is an open concern for our approach, applicable software techniques have been shown to mitigate memory-based side-channel attacks [51].

6 IMPLEMENTATION

In this section, we describe our prototype operating system, SEUSS OS (Figure 3), which provides a high-performance implementation of the SEUSS method within a full-feature FaaS platform architecture. SEUSS OS is designed to run natively on the backend compute nodes of a FaaS platform. The following is a high-level overview of the design and implementation:

- Our SEUSS OS prototype is x86_64 multicore kernel, built from the EbbRT [35] framework, that runs within a KVM-QEMU virtual machine.
- The unikernel stack of a UC is implemented using Rumprun [5], an existing port of Python or JavaScript, and an OpenWhisk invocation driver.
- UCs execute in user mode (ring 3) with page-table based hardware protection.
- A network layer in SEUSS OS masquerades traffic going in and out of UCs, allowing for external TCP connections initialized from within the guest functions.

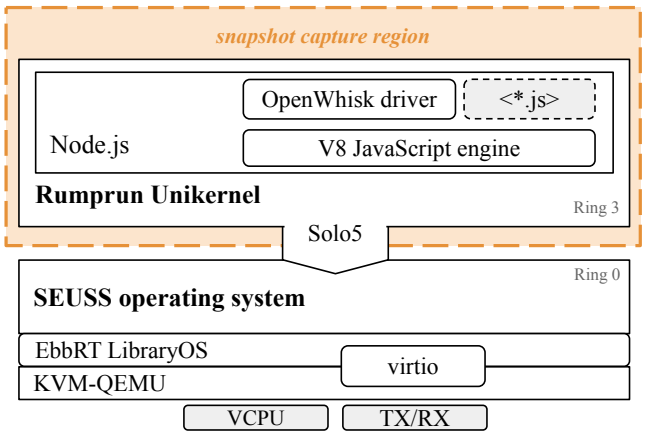

Figure 3: Single-core vertical slice of the software stack of our SEUSS OS prototype

Software Stack

SEUSS OS is written in C++ and extends EbbRT [35] library OS framework, totaling 7,971 lines of new code [1]. EbbRT provides the bottom-most software layer, On top of EbbRT, we have implemented the functionality to deploy and multiplex UCs, capture snapshots, and route UC network traffic to the external network.

SEUSS OS runs in protection ring 0 (kernel mode), while UC instances execute in protection ring 3 (user mode). Internal to every UC is the Rumprun [5] unikernel linked with a port of Node.js or Python. This version of Rumprun had been previously ported to run in userspace on top of `Solo5` [4] middleware library, which defines the minimal set of domain-crossing hypercalls.

An important requirement of SEUSS is that it supports a full set of high-level language interpreters. In our implementation, we adopt a *general-purpose* unikernel [22] as the foundation of a UC. This unintuitive design choice presents a stark contrast to the popular notion of unikernels, which are associated with low-millisecond boot times and tiny memory footprints [11, 23, 32, 33]. A general-purpose library OS will incur longer boot times and larger memory overheads, but provides wide support for the various interpreters of a serverless platform [2].

EbbRT provides the majority of low-level functionality that SEUSS OS requires (e.g., a multicore event-driven runtime, a `virtio` paravirtualized NIC, and a zero-copy TCP/IP network stack). Rumprun provides a POSIX-like unikernel based on the NetBSD source with a common set of shared system libraries and a ramdisk filesystem. Compared to other highly-specialized unikernels, Rumprun is readily able to support language runtimes used in our prototype. Solo5 made our job of supporting Rumprun unikernel instances on SEUSS OS far simpler through its minimal set of hypercalls and its use of cooperative scheduling and poll based IO.

Snapshots & Snapshot stacks

SEUSS OS uses direct access to hardware page tables to capture snapshots, deploy UCs, and to enable page-level sharing across snapshot stacks and UCs.

Triggering Snapshots. In our prototype, we use the `x86` debug register to trigger the creation of a snapshot. When the exception occurs, execution switches into a kernel mode handler that records the state of the (interrupted) UC into a new snapshot object. When finished, execution transitions back into user mode and continues transparently to the running UC. Through this method, we can pinpoint the exact instruction within the unikernel where the snapshot is captured. This technique proved invaluable when exploring various *anticipatory optimizations* (§7).

[1] Generated using David A. Wheeler's 'SLOCCount' [44].

[2] If engineering resources allow for building highly-specialized unikernels per interpreter, SEUSS snapshotting and associated optimizations can still be leveraged.

Capturing Snapshots. SEUSS OS keeps snapshots lightweight by only capturing the pages modified since the UC was created. To achieve this, we use traditional hardware semantics to track recently changed pages via the x86 dirty bits. Upon snapshotting, the complete page table structure is captured but only the dirty pages are cloned. Snapshots also retain the identification of the base snapshot that the captured UC was spawned from, which can then be used to resolve faults across later UCs. Depending on the semantics of a page fault, SEUSS OS may allocate a new page, clone a page from within the backing snapshot stack, or resolve the fault with a read-only mapping to a page within the source snapshot stack. Given these semantics, a snapshot can only be deleted safely when no other snapshots or UCs depend on it. We address this concern in our prototype by only deleting *function-specific* snapshots that have no active UCs.

Deploying UCs. The procedure of deploying function execution from a snapshot starts with creating a new UC, which includes a shallow copy of snapshot page table structure. Next, the root of the new UC page table is mapped to the core and the TLB is flushed. Execution switches into the UC and begins at the instruction where the snapshot was triggered. Execution begins by triggering a breakpoint exception and overwrite the exception frame with the register values contained within the snapshot. Upon return, the unikernel is running.

Memory Management. One issue with extensive use of copy-on-write is that memory becomes highly overcommitted and application demand may cause the system to run out of physical memory. This is a problem, for example, with the heavy use of fork() in Linux, whereby an OOM daemon may be triggered and may kill system critical processes. This is not a problem in our design, because UCs for function invocations are transient and can always be killed by the system without impacting the system's ability to make forward progress. While more complexity may be necessary in the future, our OOM daemon for SEUSS is trivial; we reclaim idle UCs that do not currently host a live invocation as soon as the available physical memory drops below a pre-defined threshold.

Networking

Each UC is configured with an identical IP and MAC address, enabling it to be trivially re-deployed across time or in parallel across cores. A default UC configuration also creates the potential to migrate snapshots across machines (§9).

A network layer monitors traffic going in and out of the UC and enables internal and external communication by forwarding and masquerading traffic down through the SEUSS OS network stack. The internal network allows the SEUSS OS invocation procedure to communicate with the running unikernel (e.g., to send the input arguments to the invocation driver). The external network proxy monitors the unikernels' access to the outside world.

A per-core *network proxy* maintains mappings for both the internal and external networks for each unikernel instance active on that core. Incoming traffic is screened, and the traffic destined for unikernels is sent through an additional translation process to determine the worker core where the UC is resident. TCP destination ports act as the unique key for mapping packets to an active UC. We currently do not support port mapping of UDP or IPv6 packets, but the approach would be similar. This design only supports outgoing TCP connections initiated from within the unikernel.

FaaS Platform Integration

To preserve full functionality with the FaaS platform, we have designed SEUSS OS to act as an OpenWhisk protocol compliant drop-in replacement for Linux, while keeping all other platform functionality unaffected. To accomplish this, we have built an intermediate shim process, written in C++ and run on Linux. The shim is responsible for reading requests from the OpenWhisk message bus (Kafka) and translating them to internal messages which are sent to a SEUSS OS VM. The advantage of this approach is that it avoids the need to support a platform-specific protocol within our lightweight OS implementation of SEUSS. The disadvantage is that it adds an additional network hop to the packet processing path. The shim processes can be written in any language to take advantage of the wealth of existing client libraries for the services that make up the FaaS platform (e.g., Kafka, CouchDB, ZooKeeper).

7 EVALUATION

We evaluate our SEUSS OS prototype (§6) compared to Linux across a series of micro and macro benchmark experiments.

Experimental Infrastructure. For our evaluations, we use a four-node cluster of physical machines connected via a 10GbE network on a private VLAN through a commodity switch. Each machine contains two 8-core Intel Xeon E5-2660 processors (16 cores in total) running at 2.20 GHz, and a Solarflare Communications SFC9120 Ethernet card. The processors have been configured to disable Turbo Boost, hyper-threads, and dynamic frequency scaling. For software, we use Ubuntu *18.04* LTS (Bionic Beaver), Linux *v4.15.0*, Docker *v18.09*, and OpenWhisk *v0.9.0*.

We deploy the FaaS compute node within a VM to maintain an identical hardware configuration for both the SEUSS OS and Linux comparisons. One physical machine acts as the dedicated host for a single `qemu-kvm` VM instance. The VM is configured with 16 VCPUs (1:1 with the host), 88 GB of memory, a virtio/vhost paravirtualized network device, and

Rumprun Unikernel	Snapshot Size (MB)	Size After AO (MB)
Node.js Invocation Driver	109.6	114.5
JavaScript NOP function	4.8	2.0
Invocation (after AO)	**Latency (ms)**	**Memory Footprint (MB)**
Cold Start:	7.5	2.05
Warm Start:	3.5	1.53
Hot Start:	0.8	0.05

Table 1: SEUSS Microbenchmarks. Top: Memory footprint of snapshots before and after Anticipatory Optimization (AO). Bottom: Invocation latency and memory footprint of NOP JavaScript functions averaged across 475 invocations.

an in-memory (ramdisk) filesystem. When running Linux within the VM, the software stack is identical to the Linux host.

Microbenchmarks: SEUSS Internals

We evaluate the performance of the SEUSS primitives for deploying a serverless function (Table 1). Opposed to evaluating complex real-world functions, we run a simple JavaScript NOP function that minimizes the time spent in the application, allowing us to focus on system-induced overheads. The results in this section are captured after we apply the anticipatory optimizations discussed in §7.

Invocation Latency. Table 1 presents the latency to deploy the NOP JavaScript function via a cold, warm, and hot invocation path (as described in §4). For the invocation latency, we measure from the moment the invocation request is received by the SEUSS OS node to the moment the function has finished executing and the result is returned from the UC back to SEUSS OS. External network latencies and FaaS control plane overheads are not included in this duration.

The cold invocation (7.5 ms) includes the time to construct and deploy the UC from the runtime snapshot, set up a TCP connection with the UC, pass in and compile the NOP function code, capture the function snapshot, pass in an empty invocation argument and execute. Capturing the NOP function snapshot (2 MB) took around 400 µs. The function ran for roughly 0.5 ms. The warm invocation (3.5 ms) deploys from the function snapshot and, therefore, includes only the overhead to connect to the UC, pass in the invocation arguments, and run. In the hot invocation (0.8 ms), an idle UC is reused, removing the overhead of UC creation and warming the internal UC pathway. From this data, we see that the time to import and compile even a single-line NOP function (roughly 5 ms of the cold start) is the largest overhead. This overhead will grow in proportion to the code size of the function being run, making warm and hot starts even more beneficial.

Memory Footprint. Table 1 presents the base snapshot sizes for the Node.js unikernel (114.5 MB) and the NOP function snapshot (2 MB), which acts as a page-level `diff` of the Node.js snapshot. Even for a NOP function, hundreds of pages are touched while importing and compiling the code. The bottom right of Table 1 presents a consequential number of pages copied during the execution of each of the three invocation types. This highlights a limitation of a page-based COW approach at a fine granularity; It is likely that far less state was generated but spread across many page boundaries. We plan to consider the runtime effects of COW on a complex function workload in future work.

	No AO	Network AO	Network + Interpreter AO
Cold Start	42 ms	16.8 ms	7.5 ms
Warm Start	7.6 ms	5.5 ms	3.5 ms

Table 2: Latency improvements across different AO

Anticipatory Optimizations. A critical part of our approach is a series of anticipatory optimizations (AO) that aim to accrue useful execution state within the base runtime snapshot. The way we achieve this is simple: before taking the base environment snapshot we warm the internal pathways and data structures of the software stack by using semantic knowledge of our common-path operation. Namely, that each function execution involves importing code via the network and compiling it before it can be run. Following this high-level operation, we employ two levels of AO. First, we send an HTTP request over the unikernel's network prior to taking the base environment snapshot (Network path in Table 2). Second, we send a "dummy" JavaScript function into the unikernel which is interpreted and run prior to taking the base environment snapshot (Exec path + Network path in Table 2).

Exercising the network stack and interpreter moves significant first-time execution off the critical path. Pre-exercising the network paths alone reduces cold start runtimes from 42.0 ms to 16.8 ms. Pre-executing both the kernel IO pathways and the state generated by the JavaScript interpreter while executing function code brings cold start times down to 7.5 ms, close to warm start times before AO, and warm start times drop from 7.6 ms to 3.5 ms. Reducing the performance gap between cold and warm starts is critical for when cold starts dominate the workload (§7).

Our approach of pre-executing likely paths prior to capturing the shared snapshot helps factor memory out of the function snapshots. As shown in Table 1, AO was able to halve the memory footprint of a NOP JavaScript function snapshot from 4.8 MB to 2.0 MB (conversely AO bloats the Node.js base runtime snapshot by 4.9 MB). As a result, we are able to double our snapshot cache size from 16, 000 to 32, 000 NOP function snapshots in our FaaS throughput macro evaluations (§7).

Microbenchmarks: Function Caching

In this experiment, we compare the overheads of SEUSS to standard system-level techniques used for isolating function

Isolation Method	Creation Rate (per second)	Cache Density
Firecracker microVM	1.3	450
Docker w/ overlay2 fs	5.3	3000
Linux process	45	4200
SEUSS UC	128.6	54000

Table 3: Cache density limit and parallel (16-way) creation rate for the Node.js runtime environments on a 88GB, 16 CPU virtual machine. The SEUSS prototype can deploy UCs quickly because deployment consists mainly of a memory copy of page table structures.

executions (i.e., processes, containers, and VMs). We consider both the latency to create new isolated instances and the memory footprints for idle instances held in memory.

Methodology. Each execution context consists of a Node.js JavaScript interpreter that is running the OpenWhisk invocation driver. Upon initialization, each instance sits blocked on a port awaiting a new connection (no code has been imported yet). We evaluate this Node.js context deployed in the following manner: standard processes, Docker containers, lightweight VMs, and SEUSS UCs.

Processes: As processes provide insufficient isolation, the purpose of this result is to show the baseline memory sharing and startup latency of Node.js on Linux.

Containers: Using the (Alpine Linux) Node.js Docker container image with the overlay2 storage driver.

microVMs: Using the Kata Container backend for Docker, we deploy the same Node.js container image within a dedicated `Firecracker` VM [6].

Cache Density. The number of Node.js instances that can sit idle on a node is important from the perspective of caching interpreters (allowing function code to be quickly be imported) and from the perspective of functions blocked on external IO. We determined the maximum number of Node.js environment instances that our virtual compute node can support by sequentially deploying instances until the memory of the VM is saturated.

As shown in Table 3, we are able to deploy around 4200 Node.js processes within our VM. When isolated in individual Docker containers, the total number of Node.js instances drops to around 3000. Unsurprisingly, the use of a container isolated within a virtual machine (with its own Linux kernel) results in an increase of over 100 MB to the per-instance memory footprint, thereby reducing the number instances that our VM could support to around 450. In comparison, SEUSS is able to deploy over 54, 000 Node.js UCs. This density is enabled by the high-degree of redundancy between the identical UC instances, which we avoid through sharing with the snapshot images.

Creation Rates. To enable a performant caching strategy, the techniques employed to create new execution contexts should be both fast and scalable. The creation latency is critical when re-populating the cache with new instances, especially when the system is under heavy load. To observe the maximum rate of instance creation, we deployed new instances in parallel across all 16 cores and measured the time it took to reach the previously observed density. Table 3 presents the results.

With processes, it took 93 s to saturate the node, resulting in an average rate of 45 instances created per second. This creation rate dropped to 5.3 instances per second when deploying Docker containers in parallel. Upon investigation, we observe container creation has two distinct scalability issues. First, the creation latency for an individual container is proportional to the number of total container instances active in the system (an observation that has been corroborated by others [33]). In our sequential deployment (density test), the creation time for a single Node.js container increased linearly from 541 ms (with no other containers on the system) to averaging 1.5 s when over 1000 containers. Second, creation latency also suffers relative to the number of parallel creations taking place at the time. In our parallel creation test, the creation times is proportional to the number of concurrent creations taking place, resulting in an average creation latency of 8.5 s when deploying container instances across all 16 cores.

When deploying lightweight virtual machines, the minimal latency to deploy a single Node.js instance grew to over 3 seconds, due to the requirement to boot the Linux kernel prior to deploying the container and runtime. This resulted in a creation rate of 1.3 instances per second.

With SEUSS, we were able to construct Node.js UCs at an average rate of 128.6 per second, a rate that is almost 2.4 times faster than Linux processes. As we will show in the following experiment, the internal mechanisms of SEUSS OS for deploying a UC imply a much faster creation rate than this. The rate we present here includes the time for the SEUSS OS shim process to communicate an invocation request over the network to the VM. While this design reflects the deployment path within a FaaS environment, the single TCP connection used by our shim process acts as a bottleneck limiting the maximum creation rate.

Microbenchmarks: Discussion. This evaluation demonstrates that the SEUSS technique is a favorable strategy for caching function execution state as snapshots. Active UCs and snapshots enable the rapid deployment of function execution while consuming little memory due to our aggressive reduction of redundancies at the page-level. In comparison, the Linux-based isolation techniques we evaluate have significant creation times and scalability limits that are especially problematic when considering the highly-parallel and latency-sensitive requirements of serverless computing.

To conclude the micro benchmark evaluation, we discuss the expected best and worst invocation overheads induced by the two deployment strategies explored in our macro benchmark experiments (§7): SEUSS snapshots vs Docker containers. In the optimal case, function execution can begin in just a few hundred *microseconds* using an idle UC or container that has been fully initialized to the particular function. SEUSS has the added option of deploying a new UC from a function-specific snapshot. The next best option for Linux is to use an idle runtime cached within a container and then import the function code. If a container does not already exist, a new one must be created. In this worst case, we observed Node.js container creation to take on average between 1.5 ms to 8.5 s, depending on the load of the system. In comparison, deploying from a runtime snapshot is a sub-millisecond operation. In addition to being much faster, the use of snapshots is far more scalable than containers, as many UCs can be deployed in parallel from a single snapshot image whereas a container is occupied for the duration of an invocation. Container occupancy is especially consequential when dealing with sudden bursts of requests while under load §7.

We are not trying to imply that containers are intrinsically more expensive for this use case, but simply that they have not yet been optimized for the large-scale and low-latency enabled by FaaS. However, since the container abstraction is spread across many different subsystems of the Linux kernel, optimizing their performance to this degree will likely involve many engineering challenges.

With VMs, we do not observe the same scalability issues with creation times as we saw with containers. However, significantly fewer VM instances were deployed in this test due to their increased memory footprint. As we discuss in §8, techniques for page-sharing across VM instances will make them compatible with the SEUSS approach.

Macro Benchmarks: FaaS Platform Performance

In this next evaluation, we analyze the impact of SEUSS on parallel function invocations on an Apache OpenWhisk cluster. We use our SEUSS OS prototype as a drop-in replacement for Linux on the backend compute node. We developed an external benchmark tool which we use to evaluate the request latency and aggregate throughput of the platform with the two different backends.

Methodology. We dedicate two of the four physical machines to host OpenWhisk, one machine to host the benchmark, and one machine to host an HTTP server used as an external endpoint for function I/O (§7). Of the two OpenWhisk machines, one hosts the control plane elements of the platforms (i.e., the controller, API server, message service, and internal databases) deployed within containers on the Linux host. The second OpenWhisk machine hosts the `QEMU-KVM` VM instance which runs either the Linux or SEUSS OS compute node.

Load Generation Benchmark. We've developed a custom FaaS load generation benchmark. The benchmark works in trials, with each trial consisting of three configuration parameters: *invocation count* (N), *function set size* (M), and *worker threads* (C). Each trial consists of N invocations distributed across a set of M functions, which are sent in a random order (for repeatability, the send order is pre-computed and persisted across trials). During a trial, C worker threads pull invocation requests (one at a time) from a shared work queue and issue a synchronous request to the FaaS platform API to process the invocation. The maximum number of requests in flight at a given time is at most C, at which point the benchmark will block until a request is returned.

Apache OpenWhisk Configuration. Each benchmark trial is performed on a fresh deployment of OpenWhisk that has been populated with the set of (M) user functions run by the benchmark. We have disabled all platform-enforced quotas and rate limits in OpenWhisk. We also prevent Docker containers from being paused when they are not in use (resulting in more stable performance on Linux when under heavy load). For the throughput tests (§7), we have disabled the 'stemcell' container cache, as the automatic initialization of containers hurt platform throughput when under heavy load. The 'stemcell' cache is re-enabled for the burst experiment (§7).

Linux Container Limit. On our Linux compute node, we set the cache size limit to 1024 containers. We had originally set the cache size closer to the observed limit of 3000 containers (Table 3) but found that a majority of the invocation requests processed by the platform would return an error. Upon investigation, we observed a scalability bottleneck in the virtual Ethernet mechanism and in-kernel packet processing that was being employed. The use of a virtual Ethernet means a single broadcast packet (e.g. ARP, DHCP) sent over a bridge interface with N connected endpoints must be processed in the kernel N separate times [46]. With 3000 endpoints, the result was a high rate of dropped packets on the bridge, causing the TCP connections between the controller process and the invocation server within the containers to timeout. Even with 1024 containers—the default limit of endpoints on a Linux bridge—we still witness connections failures during parellel invocation processing (§ 7).

Platform Throughput. This experiment is designed to stress the FaaS platform's ability to effectively cache and deploy an increasingly large set of unique functions. We consider a function 'unique' when it requires individual isolation (i.e., associated 1:1 with a distinct client account). In each trial of

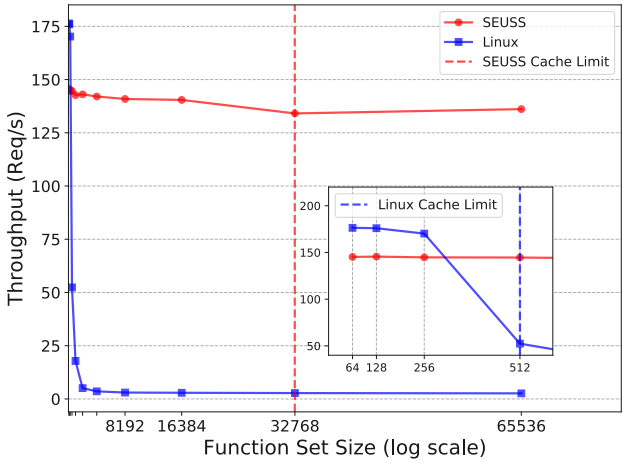

Figure 4: OpenWhisk Platform Throughput. X-axis shows the set size of functions being invoked.

Figure 5: OpenWhisk end-to-end request latency of a NOP JavaScript function across three function set sizes. Graph shows the 1st, 25th, 50th, 75th, 99th percentiles and the mean latency (dot). Note of the large difference in Y-axes ranges.

the experiment we double the set size of unique functions being invoked. While each function is logically unique, the actual code being run is the same JavaScript NOP evaluated in our microbenchmarks (§7). We chose a NOP function to stress the system-induced overheads by minimizing the time spent on the client.

Figure 4 presents the achieved throughput of OpenWhisk using a SEUSS OS compute node and Linux compute node. Trials are represented as points along the x-axis. For each trial, we doubled the number of unique functions that the benchmark invokes (ranging from 64 to 65536). The benchmark sends a continuous stream of invocation requests from 32 threads until the measured throughput reaches a point of stability.

Soon after we increase the set size of functions the Linux cache becomes saturated. At this point hot and cold invocation latencies spike and platform throughput plummets. On the mostly unique workload (right-most points on the plot), the SEUSS OS has up to a 52× speedup over Linux. The advantage here is that a cold invocation on SEUSS OS begins from an initialized Node.js snapshot as opposed to a container creation. Conversely, when the cache is saturated on Linux, every cold start requires both a cache eviction (container deletion) and a new container creation. The comparable difference in cold and warm path invocations when the cache is under-utilized (64 functions) and oversaturated (2048 functions), can be seen in Figure 5.

At low cache utilization (seen in the subplot of Figure 4) both platforms perform comparably well as the vast majority of requests get "hot" invocations. In the trials with the smallest set sizes (the left-most points on the plot), the throughput of Linux is 21% higher than that of SEUSS OS. This is because the Linux node provides better hot start latency, as our prototype introduces an additional network hop between the *shim process* and the VM (§6), which adds about 8 ms to the round-trip latency. Important to note that this is a shortcoming of our implementation and not inherent to our method.

These macro benchmark evaluations (§7) confirm that SEUSS can result in radically faster FaaS deployments (especially for cold starts), and dramatically denser function caching that enables warm and hot start invocations. In the face of increasing function set size, the SEUSS OS node is able to sustain high throughput long after the Linux node hits saturation. SEUSS radically outperforms Linux when deploying an uncached function, as the time to deploy from a snapshot is orders of magnitude faster than constructing a container. In addition, cached functions (i.e., hot starts) are often processed faster on SEUSS as Linux is quickly bogged down with container construction overheads. The container cache is primarily limited by the bridged network shared across containers. While the bridge bottleneck can likely be addressed with software changes, we believe the fundamental limitations of page sharing across containerized processes will prevent any container-based caching strategy from reaching the density achieved by SEUSS. In the next experiment we examine how high-density caching enables high-demand workloads that are otherwise unsupported by the platform.

Platform Resiliency to Request Bursts. In this experiment, we evaluate the platform's performance when exposed to the sudden arrival of invocation requests. Instead of NOP functions, we aim to simulate a more typical FaaS workload by mixing CPU-heavy functions with functions that block on

external I/O. As part of the experiment, we expose the system to a continuous stream of blocking IO functions, keeping the platform at a stable point of moderate utilization. On top of this stream, we issue a series of concurrent invocation *bursts* sent at a fixed frequency. The invocations within a burst are all made to the same CPU-bound function (functions is unqiue across bursts) with each burst simulating a compute-intensive workload triggered by a single application. The purpose of this approach is to observe when and if the background workload is affected.

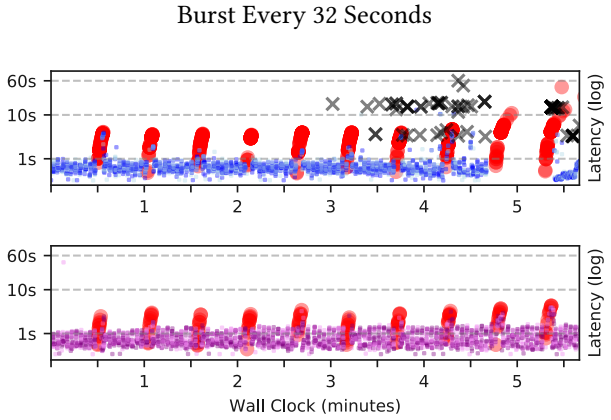

Figure 6: Request burst sent every 32 seconds; Top: Linux. Bottom: SEUSS; Dots represent individual requests: x-axis is request time sent, y-axis is request latency (log scale). Failed requests are marked with an x.

Experiment Setup. To generate the background utilization stream, we deploy our benchmark using 128 threads that make requests to a total of 16 unique IO-bound functions. The benchmark is rate-throttled to a limit of 72 requests per second [3]. Each IO-bound function makes an external network call to a remote `HTTP` server, which blocks for 250 ms before sending an `OK` reply (the function finishes once the reply is received). The CPU-bound burst functions each perform a computation that takes around 150 ms. Bursts are sent at a fixed frequency of every 32, 16, or 8 seconds. On the Linux node, we configure the OpenWhisk 'stemcell' cache to 256 Node.js containers. This container cache is beneficial for handling the sudden bursts of never-before-seen invocations. At the same time, the overheads of constructing these background containers directly competes with the container cache limits and interferes with cold start times.

Burst Resiliency on Linux . At the slowest burst frequency, 32 s between bursts (Figure 6), the Linux container cache is repopulated between bursts. In this case, the sudden arrival of requests are quickly serviced by initialized environments. However, beginning around the 5th burst, the container cache limit is hit and some of the requests begin to error (marked in the figures with a '**x**'). When the burst frequency is increased to 16 s (Figure 7) and 8 s (Figure 8), the container cache does not have time to repopulate between bursts, therefore, only the first burst is handled with low latency. When a stemcell container is unavailable, the invocations begin to see "cold start" overheads of between 10 s and 60 s. More concerning, the reliability of the Linux node suffers dramatically as the container cache becomes fully saturated. Across all three burst frequencies, requests begin to time-out and error once the container cache limit is reached. As evidenced by the gaps in the background stream of the graphs, there are periods of times where the Linux node gets overwhelmed and stops processing requests all together.

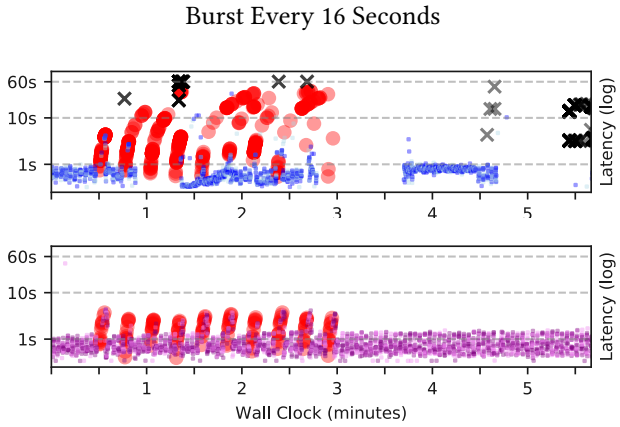

Figure 7: Request burst sent every 16 seconds; Top: Linux. Bottom: SEUSS; Dots represent individual requests: x-axis is request time sent, y-axis is request latency (log scale). Failed requests are marked with an x.

Burst Resiliency on SEUSS. With the SEUSS OS node, OpenWhisk handles every request across all three burst frequencies (no requests return an error). The background stream sees reliably lower latency on SEUSS than on Linux, and is far less disturbed by the increased arrival rate. Only at the highest burst frequency (Figure 8) can we observe a disturbance in the background stream as the CPUs become contented by the high request concurrency (and further exacerbated by EbbRT's non-preemptive event model [35]). As each burst adds only one additional snapshot to the cache, we would presumably require tens of thousands of bursts before there would be any cache contention on the SEUSS OS node. As shown in Figure 4, SEUSS OS provides high-throughput even for an all-cold invocation workload.

[3] About 40% of the container cache is consumed by the background stream on Linux.

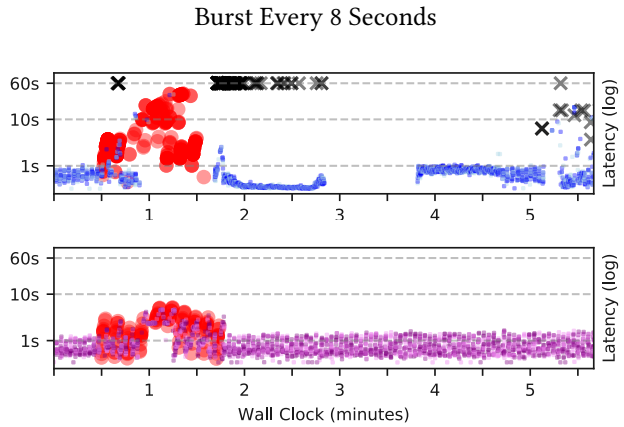

Figure 8: Request burst sent every 8 seconds; Top: Linux. Bottom: SEUSS; Dots represent individual requests: x-axis is request time sent, y-axis is request latency (log scale). Failed requests are marked with an x.

8 RELATED WORK

Recent research systems have explored the use of computational caching as a way to accelerate FaaS computation. Replayable Execution [39] uses process-level checkpoint/restore to deploy pre-warmed JVM instances within a running container. Unlike SEUSS, their techniques only target a base runtime checkpoint and do not extend to function-specific caching. SOCK [15] and SAND [19] demonstrate both shortened deployment time and decreased memory footprints by forking execution from existing interpreter processes and applying COW sharing across their address spaces.

Our approach differs from process-fork based techniques in a fundamental way. SEUSS snapshots treat the unikernel as a black box, enabling the *external* capture of arbitrary unikernel environment state. In comparison, forking requires careful coordination between the system and the user-level process being forked. In addition, a process-fork approach is limited to interpreters that internally fork (Node.js—one of the most popular runtimes for serverless—does not support POSIX fork [2]). Furthermore, process-level isolation is insufficient for mutually-untrusted functions, therefore, any process-based solution must include an additional approach for isolation or a relaxation of the security model. For example, migrating forked processes to new containers as part of the deployment procedure, or restricting forked processes to a single client (i.e., a single container). As we show in our evaluation, Linux containers introduce considerable scalability bottlenecks when used for isolating individual function executions (§7).

The techniques underlying SEUSS have been used by previous research systems for use cases extending beyond serverless computing. Library operating systems (unikernels) have been explored in the context of edge-computing [30], network function virtualization [24, 50], application sandboxing [9, 18, 36], and lightweight virtual domains [23, 31, 33, 45]. KylinX [48] explores fork-like techniques on unikernels and reduces memory footprints through dynamic page mappings. For virtual machines, Kaleidoscope [12], Potemkin [38], and Tardigrade [29] have explored fast fork-like VM cloning and copy-on-write memory sharing. Process-level checkpointing has been explored in the context of process migration [1], fast re-initialization [41], and replay debugging [37].

9 FUTURE WORK

This work leverages computational caching to achieve fast function starts, higher density function caching, and support for request bursts on a FaaS platform. Future FaaS platforms will continue to require these properties but at a scale and degree of parallelism that far exceeds that of a single compute node. We view the natural evolution of SEUSS as spanning across nodes to provide a *distributed & replicated* global cache[4]. The read-only and deploy-anywhere properties of unikernel snapshots suggest they can be cloned and deployed across machines with similar hardware profiles. A distributed SEUSS would enable advanced sharing techniques to speed up remote deployments, such as VM state coloring [12] or on-demand paging [26].

A computational caching framework allows us to study more aggressive anticipatory optimization approaches. For example, we are exploring the use of continuous hardware tracing along with machine learning to automatically identify optimization opportunities within snapshots. As in our prior work, we are looking at how cached snapshots can be synthesized from a combination of historical data and prediction mechanisms [42].

In our SEUSS OS prototype, we adopted Rumprun, a general-purpose unikernel, as the foundation of our UC. Its POSIX-like interface greatly reduced development effort because the interpreters we targeted could run out-of-the-box without the effort that would be necessary when porting them to a specialized unikernel. Going forward, we are transitioning to running applications on UKL (Unikernel Linux [34]), which targets upstream adoption into the Linux kernel for long-term maintainability. Towards the goal of increased security and broadened applicability for SEUSS, as future work we plan to explore deploying UKL-based functions onto a specialized KVM monitor.

10 CONCLUSION

At a high-level, SEUSS effectively segregates the computational state of an application into two parts: common state

[4]We will then be obliged to rename the method to *DR-SEUSS*.

which has long term value in its ability to eliminate redundant computation, and execution-specific state which is ephemeral and discardable. Our results demonstrate that this separation can be used to dramatically improve the performance of serverless function execution.

In SEUSS, unikernel snapshots speed up invocations by deploying functions within fully-initialized environments. By applying page sharing across the entire software stack, the memory footprint of snapshots are considerably smaller than processes, containers or microVMs, enabling a greater number of function instances to be cached on a node.

The techniques employed by SEUSS are simple enough that they should be easily adopted within production-grade operating systems and hardware-enforced encrypted sandboxes. Furthermore, our experience suggests that the approach explored in this paper can provide powerful benefits to applications outside of the serverless paradigm.

By combining the benefits of low latency cold starts and simplified immutable memory caching, SEUSS is able to handle sudden request bursts that are currently unsupported by the traditional approach. These results show that the serverless function need not be considered a heavyweight computational primitive. When highly-parallel executions can be deployed in milliseconds, serverless functions will support general-purpose computing across arbitrary scales.

Acknowledgments: This work would not have been possible without the significant efforts of others. We would like to thank Frank Bellosa, our shepherd, Kyle Hogan, and Andrea Burns for their help in preparing the final version of this paper. Thanks to Dan Williams and Ricardo Koller for the unikernel discussions and efforts on the Solo5 project. Special thanks to Devosh Mathivanan for his work developing the SEUSS benchmark. Finally, we owe a great deal of gratitude to the Red Hat Collaboratory@BU and the industry partners of the Mass Open Cloud (MOC), including Red Hat, Two Sigma, and Intel. This work was supported by the National Science Foundation under award IDs CNS-1254029, CCF-533663, and CNS-1414119.

Open Source: Project code and documentation can be found at https://github.com/SESA/SEUSS.

REFERENCES

[1] (Accessed 3/30/2020). *Checkpoint/Restore In Userspace.* https://www.criu.org

[2] (Accessed 3/30/2020). Node.js v13.11.0 Documentation. https://nodejs.org/api/child_process.html

[3] (Accessed 3/30/2020). Seccomp security profiles for Docker. https://docs.docker.com/engine/security/seccomp/

[4] (Accessed 3/30/2020). *Solo5: A sandboxed execution environment for unikernels.* https://github.com/Solo5/solo5

[5] (Accessed 3/30/2020). *The Rumprun unikernel.* https://github.com/rumpkernel/rumprun

[6] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight Virtualization for Serverless Applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 419–434. https://www.usenix.org/conference/nsdi20/presentation/agache

[7] Jonathan Appavoo and Amos Waterland. 2008. Exposing and Exploiting Structure in Computation: A Unification Principle of Information Processing Systems. http://www.cs.bu.edu/~jappavoo/Resources/Papers/PSML/dsrc2.pdf DSRC/DARPA Summer Conference.

[8] Andrea Arcangeli, Izik Eidus, and Chris Wright. 2009. Increasing memory density by using KSM. In *Proceedings of the linux symposium*. 19–28. http://www.kernel.org/doc/ols/2009/{%}5Cnhttps://www.kernel.org/doc/ols/2009/ols2009-pages-19-28.pdf

[9] Andrew Baumann, Marcus Peinado, and Galen Hunt. 2015. Shielding Applications from an Untrusted Cloud with Haven. *ACM Trans. Comput. Syst.* 33, 3, Article 8 (Aug. 2015), 26 pages. https://doi.org/10.1145/2799647

[10] Zack Bloom. 2019. Cloud Computing without Containers. https://blog.cloudflare.com/cloud-computing-without-containers/

[11] A. Bratterud, A. Walla, H. Haugerud, P. E. Engelstad, and K. Begnum. 2015. IncludeOS: A Minimal, Resource Efficient Unikernel for Cloud Services. In *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*. 250–257. https://doi.org/10.1109/CloudCom.2015.89

[12] Roy Bryant, Alexey Tumanov, Olga Irzak, Adin Scannell, Kaustubh Joshi, Matti Hiltunen, H. Andres Lagar-Cavilla, and Eyal de Lara. 2011. Kaleidoscope: Cloud Micro-Elasticity via VM State Coloring. In *European Conference on Computer Systems (Eurosys)*. Saltzburg, Austria.

[13] N. Di Pietro and E. Calvanese Strinati. 2019. An Optimal Low-Complexity Policy for Cache-Aided Computation Offloading. *IEEE Access* 7 (2019), 182499–182514. https://doi.org/10.1109/ACCESS.2019.2959986

[14] Xianzheng Dou, Peter M. Chen, and Jason Flinn. 2019. ShortCut: Accelerating Mostly-Deterministic Code Regions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 570–585. https://doi.org/10.1145/3341301.3359659

[15] Edward Oakes and Leon Yang and Dennis Zhou and Kevin Houck and Tyler Harter and 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

[16] D. Gruss, M. Lipp, M. Schwarz, D. Genkin, J. Juffinger, S. O'Connell, W. Schoechl, and Y. Yarom. 2018. Another Flip in the Wall of Rowhammer Defenses. In *2018 IEEE Symposium on Security and Privacy (SP)*. 245–261. https://doi.org/10.1109/SP.2018.00031

[17] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C. Arpaci-Dusseau, and Remzi H.

Arpaci-Dusseau. 2016. Serverless Computation with OpenLambda. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*. USENIX Association, Denver, CO. https://www.usenix.org/conference/hotcloud16/workshop-program/presentation/hendrickson

[18] Jon Howell, Bryan Parno, and John R. Douceur. 2013. Embassies: Radically Refactoring the Web. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. USENIX, Lombard, IL, 529–545. https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/howell

[19] Istemi Ekin Akkus and Ruichuan Chen and Ivica Rimac and Manuel Stein and Klaus Satzke and Andre Beck and 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

[20] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Menezes Carreira, Karl Krauth, Neeraja Yadwadkar, Joseph Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. 2019. *Cloud Programming Simplified: A Berkeley View on Serverless Computing*. Technical Report UCB/EECS-2019-3. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-3.html

[21] Jonas, Eric and Pu, Qifan and Venkataraman, Shivaram and Stoica, Ion and Recht, Benjamin. 2017. Occupy the Cloud: Distributed Computing for the 99%. In *Proceedings of the 2017 Symposium on Cloud Computing (SoCC '17)*. ACM, New York, NY, USA, 445–451. https://doi.org/10.1145/3127479.3128601

[22] Antti Kantee. 2016. Flexible Operating System Internals: The Design and Implementation of the Anykernel and Rump Kernels. *Aalto University* (2016), 218. http://www.fixup.fi/misc/rumpkernel-book/rumpkernel-bookv2-20160802.pdf

[23] Avi Kivity, Dor Laor, Glauber Costa, Pekka Enberg, Nadav Har'El, Don Marti, and Vlad Zolotarov. 2014. OSv—Optimizing the Operating System for Virtual Machines. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. USENIX Association, 61–72. https://www.usenix.org/conference/atc14/technical-sessions/presentation/kivity

[24] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. 2000. The Click Modular Router. *ACM Trans. Comput. Syst.* 18, 3 (Aug. 2000), 263–297. https://doi.org/10.1145/354871.354874

[25] 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

[26] Lagar-Cavilla, Horacio Andrés and Whitney, Joseph Andrew and Scannell, Adin Matthew and Patchin, Philip and Rumble, Stephen M. and de Lara, Eyal and Brudno, Michael and Satyanarayanan, Mahadev. 2009. SnowFlock: Rapid Virtual Machine Cloning for Cloud Computing. In *Proceedings of the 4th ACM European Conference on Computer Systems (EuroSys '09)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/1519065.1519067

[27] G. Lee, W. Saad, and M. Bennis. 2017. Online optimization for low-latency computational caching in Fog networks. In *2017 IEEE Fog World Congress (FWC)*. 1–6.

[28] Jens Lindemann and Mathias Fischer. 2018. A memory-deduplication side-channel attack to detect applications in co-resident virtual machines. In *Proceedings of the ACM Symposium on Applied Computing*. 183–192. https://doi.org/10.1145/3167132.3167151

[29] Jacob R Lorch, Andrew Baumann, Lisa Glendenning, Dutch T. Meyer, and Andrew Warfield. 2015. Tardigrade: Leveraging lightweight virtual machines to easily and efficiently construct fault-tolerant services. In *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2015*. 575–588.

[30] 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

[31] 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

[32] Anil Madhavapeddy and David J. Scott. 2014. Unikernels: The Rise of the Virtual Library Operating System. *Commun. ACM* 57, 1 (2014), 61–69. https://doi.org/10.1145/2541883.2541895

[33] 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

[34] Ali Raza, Parul Sohal, James Cadden, Jonathan Appavoo, Ulrich Drepper, Richard Jones, Orran Krieger, Renato Mancuso, and Larry Woodman. 2019. Unikernels: The Next Stage of Linux's Dominance. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS '19)*. Association for Computing Machinery, New York, NY, USA, 7–13. https://doi.org/10.1145/3317550.3321445

[35] Dan Schatzberg, James Cadden, Han Dong, Orran Krieger, and Jonathan Appavoo. 2016. EbbRT: A Framework for Building Per-application Library Operating Systems. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, Berkeley, CA, USA, 671–688. http://dl.acm.org/citation.cfm?id=3026877.3026929

[36] Zhiming Shen, Zhen Sun, Gur-Eyal Sela, Eugene Bagdasaryan, Christina Delimitrou, Robbert Van Renesse, and Hakim Weatherspoon. 2019. X-Containers: Breaking Down Barriers to Improve Performance and Isolation of Cloud-Native Containers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 121–135. https://doi.org/10.1145/3297858.3304016

[37] Dirk Vogt, Cristiano Giuffrida, Herbert Bos, and Andrew S. Tanenbaum. 2015. Lightweight Memory Checkpointing. In *Proceedings of the 2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '15)*. IEEE Computer Society, Washington, DC, USA, 474–484. https://doi.org/10.1109/DSN.2015.45

[38] Michael Vrable, Justin Ma, Jay Chen, David Moore, Erik Vandekieft, Alex C. Snoeren, Geoffrey M. Voelker, and Stefan Savage. 2005. Scalability, fidelity, and containment in the potemkin virtual honeyfarm. In *Proceedings of the twentieth ACM symposium on Operating systems principles - SOSP '05*. ACM Press. https://doi.org/10.1145/1095810.1095825

[39] Kai Ting Amy Wang, Rayson Ho, and Peng Wu. 2019. Replayable execution optimized for page sharing for a managed runtime environment. *Proceedings of the 14th EuroSys Conference 2019* (2019). https://doi.org/10.1145/3302424.3303978

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

CA, USA, 133–145. http://dl.acm.org/citation.cfm?id=3277355.3277369

[41] Yi-Min Wang, Yennun Huang, Kiem-Phong Vo, Pe-Yu Chung, and C. Kintala. 1995. Checkpointing and Its Applications. In *Proceedings of the Twenty-Fifth International Symposium on Fault-Tolerant Computing (FTCS '95)*. IEEE Computer Society, Washington, DC, USA, 22–. http://dl.acm.org/citation.cfm?id=874064.875647

[42] Amos Waterland, Elaine Angelino, Ryan P. Adams, Jonathan Appavoo, and Margo Seltzer. 2014. ASC: Automatically Scalable Computation. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*. Association for Computing Machinery, New York, NY, USA, 575–590. https://doi.org/10.1145/2541940.2541985

[43] Amos Waterland, Elaine Angelino, Ekin D. Cubuk, Efthimios Kaxiras, Ryan P. Adams, Jonathan Appavoo, and Margo Seltzer. 2013. Computational Caches. In *Proceedings of the 6th International Systems and Storage Conference (SYSTOR '13)*. Association for Computing Machinery, New York, NY, USA, Article Article 8, 7 pages. https://doi.org/10.1145/2485732.2485749

[44] David A Wheeler. [n.d.]. SLOCCount. http://www.dwheeler.com/sloccount/.

[45] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. 2002. Scale and Performance in the Denali Isolation Kernel. *SIGOPS Oper. Syst. Rev.* 36, SI (Dec. 2002), 195–209. https://doi.org/10.1145/844128.844147

[46] Dave Wilder. 2017. Scaling the Docker Bridge Network. https://developer.ibm.com/linuxonpower/2017/05/26/scaling-docker-bridge-network/

[47] Dan Williams, Ricardo Koller, Martin Lucina, and Nikhil Prakash. 2018. Unikernels As Processes. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC '18)*. ACM, New York, NY, USA, 199–211. https://doi.org/10.1145/3267809.3267845

[48] Yiming Zhang and Jon Crowcroft and Dongsheng Li and Chengfen Zhang and Huiba Li and Yaozheng Wang and Kai Yu and Yongqiang Xiong and Guihai Chen. 2018. KylinX: A Dynamic Library Operating System for Simplified and Efficient Cloud Virtualization. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 173–186. https://www.usenix.org/conference/atc18/presentation/zhang-yiming

[49] Ethan G. Young, Pengfei Zhu, Tyler Caraza-Harter, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2019. The true cost of containing: A gVisor case study. *11th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud 2019, co-located with USENIX ATC 2019* (2019).

[50] Wei Zhang, Jinho Hwang, Shriram Rajagopalan, K.K. Ramakrishnan, and Timothy Wood. 2016. Flurries: Countless Fine-Grained NFs for Flexible Per-Flow Customization. In *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies (CoNEXT '16)*. ACM, New York, NY, USA, 3–17. https://doi.org/10.1145/2999572.2999602

[51] Ziqiao Zhou, Michael K Reiter, and Yinqian Zhang. 2016. A software approach to defeating side channels in last-level caches. In *Proceedings of the ACM Conference on Computer and Communications Security*, Vol. 24-28-Octo. 871–882. https://doi.org/10.1145/2976749.2978324 arXiv:1603.05615