Sledge: a Serverless-first, Light-weight Wasm Runtime for the Edge

Phani Kishore Gadepalli
George Washington University
phanikishoreg@gwu.edu

Sean McBride
George Washington University
seanmcbride@gwu.edu

Gregor Peach
George Washington University
peachg@gwu.edu

Ludmila Cherkasova
Arm Research
lucy.cherkasova@arm.com

Gabriel Parmer
George Washington University
gparmer@gwu.edu

Abstract

Emerging IoT applications with real-time latency constraints require new data processing systems operating at the Edge. Serverless computing offers a new compelling paradigm, where a user can execute a small application without handling the operational issues of server provisioning and resource management. Despite a variety of existing commercial and open source serverless platforms (utilizing VMs and containers), these solutions are too heavy-weight for a resource-constrained Edge systems (due to large memory footprint and high invocation time). Moreover, serverless workloads that focus on per-client, short-running computations are not an ideal fit for existing general purpose computing systems.

In this paper, we present the design and implementation of *Sledge* – a novel and efficient WebAssembly-based serverless framework for the Edge. *Sledge* is optimized for supporting unique properties of serverless workloads: the need for high density multi-tenancy, low startup time, bursty client request rates, and short-lived computations. *Sledge* is designed for these constraints by offering (i) optimized scheduling policies and efficient work-distribution for short-lived computations, and (ii) a light-weight function isolation model implemented using our own WebAssembly-based software fault isolation infrastructure. These lightweight sandboxes are designed to support high-density computation: with fast startup and teardown times to handle high client request rates. An extensive evaluation of *Sledge* with varying workloads and real-world serverless applications demonstrates the effectiveness of the designed *serverless-first* runtime for the Edge. *Sledge* supports up to 4 times higher throughput and 4 times lower latencies compared to *Nuclio*, one of the fastest open-source serverless frameworks.

Keywords Edge computing, WebAssembly, serverless, IoT

Middleware '20, December 7–11, 2020, Delft, Netherlands

ACM ISBN 978-1-4503-8153-6/20/12…$15.00
https://doi.org/10.1145/3423211.3425680

1 Introduction

Serverless computing, also known as Function-as-a-Service (FaaS), has been the fastest-growing type of cloud service over the two last years [67], achieving 50% year-over-year growth in 2019 alone. This is in part due to serverless computing's new event-driven execution model, which enables users to run small stateless applications without concerns for server provisioning or resource management issues. Since the appearance of Amazon Lambda in 2014 [11], numerous cloud providers have released alternative serverless platforms, including Google Cloud Functions [31], Microsoft Azure Functions [53], IBM Cloud Functions [37], and Alibaba Cloud Functions [7]. While implementation details differ, most utilize virtual machines (VMs) or containers as a sandbox environment for hosting the tenants and executing their functions. These frameworks are somewhat heavy-weight and not efficient for providing a small memory footprint or supporting low latencies, especially when functions are instantiated for the first time. Multiple recent trends and related technological challenges motivate our interest in *resource-efficient serverless computing at the Edge*. Among them are the following:

Rapid Proliferation of IoT: IoT has introduced an unprecedented number of low-cost devices that continuously sense and generate data. The sheer volume of data produced by IoT networks is overwhelming. Harnessing the full potential of IoT requires new computing approaches along the entire data processing pipeline.

Novel Applications Relying on Real-Time Services: The surge of Industrial IoT and next generation technologies enable a growing number of novel applications relying on extremely low (10ms) latency processing, such as:

- a smart city with street and traffic lights that communicate with each other to improve the emergency response time of first responders;
- surveillance drones with real-time monitoring and intelligent video processing for diverse set of situations (latency/time critical in the emergency situations);

- connected cars and related data services that provide drivers timely alerts about dangerous road conditions ahead.

These emerging applications should process a large amount of data coming from multiple sources in near real-time. They require data processing systems with new performance characteristics: *low latency overhead* and *high system throughput*, while focusing on spatial and temporal isolation of requests from multiple tenants.

Importance of Processing at the Edge: While Cloud computing provides a good solution for applications designed with human perception in mind, it becomes inadequate for novel latency-critical applications that rely on fast, automated decisions made with no human in the loop. For example, the data analysis and control logic of Industrial Control Systems may require a response time within 10ms [16, 81] – a characteristic that cannot be met by services delivered from the Cloud. Many machine learning (ML) workloads need to run at the very edge of the physical world (where sensing and data collection take place), but resource and battery constraints remain a major technological challenge. To satisfy the performance requirements of such workloads, we must augment on-device compute with a new class of services that can process data closer to the source, i.e., Edge computing.

Below we summarize the *execution properties and requirements* for an efficient serverless runtime at the Edge:

- *Event-driven, short-lived execution.* A prominent characteristic of serverless functions is that they are event-triggered (for example, in response to a client request) and short-lived. They are often called *stateless* as they are reclaimed and started anew for each client request. Note, that the existing, traditional systems are optimized for long-running execution over many client interactions (*e.g.*, servers and daemons), and they are not well-suited for this new, atypical execution pattern.
- *High density multi-tenancy.* The Edge represents *a limited and resource-constrained environment*, and therefore Edge resources need to be carefully shared and managed between multiple tenants. Given this, multi-tenancy at the Edge will require high density. While serverless infrastructures are often optimized for scalability at the data-center scale, this paper emphasizes the ability to efficiently support this high-density on the Edge's limited resources.
- *Low latency.* The proximity of Edge to sensors, IoT, and networked devices results in low networking latency (enabled by avoiding the WAN). Therefore, efficient, low latency execution of serverless functions at the Edge gets very important (as normally-hidden computing overheads of existing infrastructures might become dominant ones compared to low network latencies).
- *High churn and repeated execution.* A very short-lived execution of serverless functions drastically increases the churn in the system. The *rate of function instantiation and destruction* is proportional to the rate of client requests. Additionally, the specific code for a function is executed repeatedly, once for each client request.

These serverless properties define a specific set of requirements that are not best provided by existing systems. Though the serverless execution properties are atypical, these are often executed on existing systems optimized for long-running executions (*e.g.*, containers, VMs, servers, and daemons).

Recently, WebAssembly (Wasm) [33] with its light-weight sandboxing capabilities has emerged as a promising approach for supporting serverless at the Edge [23, 28, 34, 70]. Since many existing Wasm runtimes exhibit significant overheads compared to application native execution [33, 34, 38], we implement our own LLVM-based ahead-of-time (AoT) Wasm compiler, called *aWsm* (pronounced as *awesome*), which offers configurable sandboxing and optimized for performance.

In this work, we propose a new *serverless-first* infrastructure *Sledge*[1] (**S**erver**L**ess at the **Edge** runtime) optimized for properties of low-latency serverless execution at the Edge. Toward this, we focus on the serverless runtimes for **single host servers** ranging from powerful multi-core servers to low-cost systems like Raspberry Pi. A new runtime *Sledge* enables light-weight function instantion and isolation facilities, and uses kernel bypass to enable specialized serverless function scheduling. The *Sledge* runtime focuses squarely on efficiency of serverless functions, and enabling strong spatial and temporal isolation of multi-tenant function executions. The efficiency of *Sledge* is enabled by our *aWsm* compiler and serverless runtime with the following mechanisms, summarizing our **contributions:**

- *Light-weight function isolation. Sledge* executes functions in light-weight Wasm sandboxes, removing the high overheads of traditional runtimes (*e.g.*, VMs and containers), while still providing strong memory isolation.
- *Optimized function startup for high densities.* Given the repeated execution of the same functions across many tenants, *Sledge* optimizes function startup by decoupling the processing (linking and loading) of function binaries from function instantiation.
- *Decoupling work-distribution from temporal isolation. Sledge* leverages the short-lived execution properties of serverless to specialize system scheduling by decoupling both the work-distribution and load balancing across cores for scalability, from the scheduling logic to maintain fairness and temporal isolation.
- *Serverless execution performance evaluation.* We perform an extensive evaluation of *Sledge* serverless runtime using various Edge workloads to show that *Sledge* provides up to 4 times better latencies and throughputs compared

[1] www.github.com/gwsystems/sledge-serverless-framework

to *Nuclio* [57], one of the fastest open-source serverless frameworks. We also evaluate our *Sledge* Wasm compiler and its runtime on x86_64 and AArch64 architectures to show an average performance overhead within 13% of the native code execution for PolyBench/C [62] benchmarks and compare its efficiency with various existing LLVM- and Cranelift-based [19] Wasm compilers and runtimes.

The rest of this paper is organized as follows: Section 2 provides the background on existing serverless technologies and outlines related work. In Section 3, after introducing WebAssembly, we present the *Sledge* compiler and describe the design of the *serverless-first Sledge* runtime. Section 4 discusses the implementation of the *Sledge* serverless runtime, while in Section 5, we present the performance evaluation of the *Sledge* compiler *aWsm* and detailed performance assessment of our serverless framework *Sledge* for various Edge workloads. Finaly, we make concluding remarks in Section 6.

2 Background and Related Work

There are two traditional approaches for serverless implementation: *VM-based* and *Container-based* frameworks.

VM-Based FaaS: Figure 1(a) reflects a serverless function execution in a VM environment. All the yellow boxes in Figure 1(a) constitute a function in a VM environment. A hypervisor offers a software layer to abstract the physical hardware from the guest operating system(s) (VMs) [21]. It logically divides and manages the underlying system resources between guest VMs. An instance of a function is a combination of its code, memory (stack and heap), its language runtime and the library dependencies. Typically, serverless functions are transient and short-lived. A function footprint is significantly smaller compared to a size of a provisioned VM (with GBs of memory). Despite a VM-like isolation being the current standard for multi-tenant hardware sharing [77], VMs are too heavy-weight for serverless function execution. There have been efforts to provide lighter-weight VMs with faster startup times than traditional VMs by Amazon Firecracker microVM [26, 27], LightVM [52], and Kata Containers [41]. Amazon Firecracker is a minimalistic VMM that uses Linux kernel-based virtual machines (KVM) with memory footprint of 5MB and startup latency of about 125ms. It is one of the enabling technology behind the leading serverless cloud platform AWS Lambda [11].

Container-Based FaaS: Containers provide a standard way to package the user application's code, configurations, and all the dependencies into a single object. They share an operating system installed on the server hardware as shown in Figure 1(b). The containers are isolated from each other using security features such as Linux chroot, control groups (cgroups), and namespaces. Linux Containers (LXC) [48], Dockers [20], and Windows Containers [83] are different container runtimes, which containerize the applications into processes with a dedicated filesystem, providing a full execution environment, while sharing the host binaries and libraries, where appropriate. Many leading cloud providers like Amazon, Google, Microsoft, IBM and others use this technology as an enabling platform for FaaS [36, 39, 77, 82, 84].

Figure 1(b) depicts a serverless function execution in a containerized environment. Each function (white boxes) is packaged into a container image with the language runtime and dependencies including libraries (yellow boxes, which are not even shared between containers of a single function), thereby impacting the memory footprint and startup latencies of each function. The blue layers are shared between different functions, and the underlying OS controls container/function scheduling in this framework. The container manager controls creation, deletion, starting, and stopping containers in the serverless environment. Despite the efficiency gains of using containers over VMs, isolating each function in its own container still implies a significant overhead.

Figure 1(c) depicts the next evolution step in implementing the serverless infrastructures by using a *process* per function, while maintaining a container per tenant, such as the latest *Nuclio* [57] and OpenFaaS [60] runtimes. This represents an interesting design point: a process created per function isolates functions from each other, while allowing them to share the container resources. By allocating a container per tenant, this approach supports CPU, memory, and namespace isolation between tenants. We will show later that our solution *Sledge* shown in Figure 1(d) is able to accommodate much of the serverless runtime safely into the same processes as functions to minimize redundant resource consumption and customize policies for function execution. Given that the *Nuclio* platform [57] demonstrates the best results among the Kubernetes-based serverless solutions considered in [46], we choose *Nuclio* as the baseline against which we compare performance of *Sledge*.

Serverless Performance Challenges: The *startup latency* for initiating a function instance in a FaaS platform can vary significantly: from a few milliseconds to several seconds. A function execution might be a "*warm-start*", reusing a VM or container from the previous event of the same function, or it might be a "*cold-start*", where a new VM/container has to be launched. Cold-start latency depends on many additional factors: the language used for programming the function, the number and types of libraries, their dependencies, etc. While many of these factors are under a developer's control and considered for optimization [9, 49, 54, 58, 73, 86], one can achieve only a limited reduction of the cold-start latency.

The other challenging issue is *selecting the right amount of resources (memory and compute)* for executing the function. Requesting the "*right*" amount of compute is not always possible in the serverless frameworks. For example, currently in AWS Lambda, the user can only provide a memory setting. The minimum allocated memory is 128MB, with 64MB increments. Once a user selects a memory size, AWS allocates the CPU quota proportionally. Multiple papers [1, 2, 42, 77]

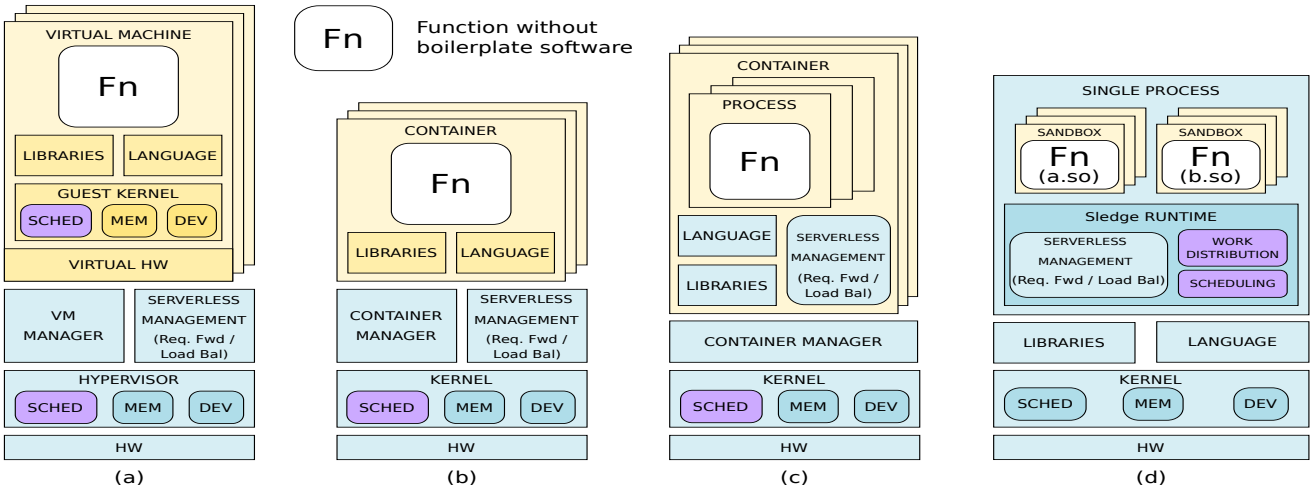

Figure 1. (a) VM-based Serverless (e.g., AWS Lambda using Firecrackers, Microsoft Azure Functions using Hyper-V, etc.), **(b)** Container-based Serverless (e.g., OpenWhisk, Google Cloud Functions, etc.), **(c)** Container + Processes-based Serverless (e.g., *Nuclio*, OpenFaaS), **(d)** *Sledge* Approach for Serverless at the Edge.

show that, while typically, function performance correlates with a memory size, it often exhibits inconsistent behavior.

Novel Approaches for Serverless Platforms: The recent popularity of serverless functions and the goal of optimizing the execution of multi-tenant workloads, brought a renewed attention to Linux kernel's abstractions [24, 43, 65]. *Unikernels* comprise a minimal operating system and a single application, making them a natural fit for serverless functions. In [43], the authors show that bypassing the kernel with unikernels can yield at least *a factor of 6 better latency and throughput*. In our *Sledge* design, we pursue a similar opportunity: bypassing the Linux kernel for efficiency, optimized latency, and a customized function scheduler.

With the advancement of IoT and novel applications requiring near real-time processing, there is a need to efficiently support a large number of serverless functions using limited hardware resources at the Edge, while delivering fast response. The latest novel approaches, aiming to support these requirements, leverage *WebAssembly (Wasm)* [33]. Wasm provides a light-weight sandboxing capabilities which can be utilized by running serverless functions at the Edge as shown by newly introduced commercial products (in 2019) from Cloudflare [70, 75] and Fastly [3, 4, 23], and research publications [14, 28, 34, 87].

Zhang, et al [87] describe Shredder, a low-latency multi-tenant cloud store that allows small units of computation (a la serverless functions) to be performed directly within storage nodes, using Javascript or WebAssembly programs in V8 for light-weight function isolation. Boucher, et al [14] suggest a novel design for serverless platforms that runs user-submitted microservices within shared processes. The paper advocates in favor of language-based (Rust) isolation, which could provide micro-second scale invocation latencies for lightweight, short-lived tasks. Hall, et al [34] demonstrate the potential of a Wasm-based approach to *significantly decrease the cold-start latencies* (under 30 ms in their implementation), while Fastly has announced that Lucet [4] (Fastly's native WebAssembly compiler and runtime) can instantiate Wasm modules in under 50 μsec [3].

In our earlier paper [28], we discuss a set of open challenges and promising opportuinities for serverless computing at the Edge. This earlier paper outlines the initial principles of the *aWsm* compiler and the related serverless runtime. Our current paper offers a comprehensive design of a fully functional system, capable of interfacing with clients. We provide the implementation details of the *aWsm* compiler and its thorough evaluation with a detailed comparison to other existing Wasm compilers. We introduce our serverless runtime implementation *Sledge*, detailing all its novel aspects and comparing its performance against the best currently existing serverless infrastructures.

Another approach with an aim to provide stateful serverless computing, Faasm [69], for big-data applications, leverages Wasm for light-weight memory isolation, and Linux's `cgroups` to provide CPU isolation for function instances executing in dedicated system-level threads. The Faasm runtime's average function initialization time is 10ms, and it relies on function snapshots, warm faaslets, and Linux kernel for execution performance. In our paper, the *serverless-first Sledge* runtime enables the optimized μ-second level function startup times by decoupling the function linking and loading from function creation and provides near real-time function execution latencies using kernel bypass.

Hall, et al [34] highlights the performance challenges related to a Wasm execution overhead compared to native application performance. Three applications used in the paper had 2-6 times higher Wasm execution overhead compared to their native execution.

A number of latest papers [33, 38] (written over period of three years) are devoted to optimize Wasm compilers and

performance of the resulting code. In 2017, only **7** out of **30** PolyBench/C [62] benchmarks performed within 1.1 times of native execution [33]. While by May 2019, **13** benchmarks out of **30** could perform within 1.1 times of native [38] due to improved Wasm compilers. In our paper, though the focus is on the *Sledge* serverless runtime, we demonstrate that our *aWsm* compiler performs within 1.1 times of the native execution for **24** out of **30** PolyBench/C benchmarks.

3 Serverless-First Runtime Design

This section presents the design of the *Sledge* serverless execution framework for the Edge. It leverages lightweight Wasm sandboxes to enable the execution of multi-tenant serverless functions with high-churn and microsecond-level latency on the resource-constrained Edge systems.

3.1 WebAssembly Background

Serverless computing at the Edge requires low-latency, high-density, a support for high-churn, and strong isolation. In order to support these requirements, the serverless-first runtime needs new means for a processes abstraction and novel fault isolation techniques.

WebAssembly (Wasm) [33] provides a strong foundation that enables a run-time to address these goals. It provides a memory-light sandbox for untrusted execution based on software fault isolation [76] and control flow integrity [5]. This sandbox enables *Sledge* to move serverless and OS services into a light-weight runtime that utilizes user-level scheduling, efficient work distribution, and asynchronous I/O to optimize for efficiency and low-latency.

Wasm is a portable, language-agnostic, low-level bytecode. Despite being driven by web standards bodies, the specification has been designed to work outside the browser as well, and there are numerous implementatons [4, 78, 79] that provide embeddings for non-Web environments.

Importantly, Wasm is a low-level language that is a compilation target for existing compilers like LLVM, that does not prescribe garbage collection. Thus, even unsafe languages such as C and C++ can compile to Wasm and execute safely. The *two components* of safe execution of unsafe languages are *memory safety* and *control-flow integrity*.

Memory-safety. In contrast to language runtimes like the JVM that rely on *type-safety* for safe program execution, WebAssembly (Wasm) [33] uses *memory-safety* that simply limits Wasm code to native loads and stores within a contiguous "linear memory". The Wasm *sandbox* ensures that *logic within the runtime cannot maliciously or accidentally corrupt memory or control outside of the sandbox*. Each Wasm memory access addresses linear memory at an *offset* from the linear memory base. Accessing bytes beyond the linear memory's size generates a sandbox violation exception. The Wasm runtime is responsible for bounds checking and the translation of linear memory accesses into the hardware's virtual addresses.

Control-flow integrity. Sandboxing untrusted execution requires control-flow integrity (CFI) [5], which constrains execution to the safe control flow graph considered by the compiler. Wasm achieves this by using the following two techniques.

First, the C execution stack holds return addresses that can be corrupted to hijack control flow using a buffer overflow. Thus, the Wasm execution stack must be *outside of linear memory*, thus inaccessible to loads and stores from within the sandbox. However, this complicates passing pointers to stack-allocated variables. Therefore, Wasm separates the execution stack into two separate stacks: (1) a stack external to linear memory, and (2) a stack within linear memory containing stack-allocated variables.

Second, Wasm requires that function pointers (including C++ vtable dispatch) cannot be de-referenced directly. Instead, calling a function pointer is translated into an activation of a function in a runtime (safe, pre-calculated) table of valid entry points and types. The runtime mediates function pointer invocations by (1) validating that they point to valid function addresses, and (2) validating that the type of the caller matches those of the function. Similarly, the entry-points into sandbox and runtime functions (that the sandbox can invoke) are indexed into the runtime function pointer table.

Wasm promises to provide the *strongly sandboxed safety of the execution of unsafe code* by appropriately constraining memory accesses and control flow.

3.2 Managing Lightweight Function Isolation and Churn using Wasm Sandboxing

The *Sledge* system leverages the Wasm security model to enable execution of multiple untrusted modules in the same process, thereby providing significantly lighter-weight isolation compared to VMs and containers for multi-tenant serverless execution. A Wasm runtime aims to maintain memory safety and control-flow integrity (CFI) by efficiently performing linear memory access bounds checks, stack operations, and function pointer invocations. The security model of Wasm allows for implicit enforcement of CFI through structured control-flow [80], thus protecting the Wasm modules from a variety of control-flow hijacking attacks including stack smashing, and return-oriented programming. However, many of the existing Wasm runtimes [35, 38] can exhibit significant overhead in properly sandboxing code. Motivated by this, we designed an ahead-of-time (AoT) compiler for Wasm to LLVM [50] bytecode and enabled configurable mechanisms for bounds checking that leverage varying levels of hardware support. We use an existing compiler (LLVM) to compile source code (*e.g.*, C/C++) to Wasm binary format. The compiled Wasm binary file is then input to our *Sledge*

compiler *aWsm*[2], generating LLVM intermediate representation (bytecode). The *Sledge* compiler is implemented in Rust and is **4,500** lines of code. We rely on LLVM to optimize for the target architecture.

Using the compilation of native C/C++ code into a sandboxed binary for *Sledge* as an example, Figure 2 demonstrates our compilation and runtime pipeline.

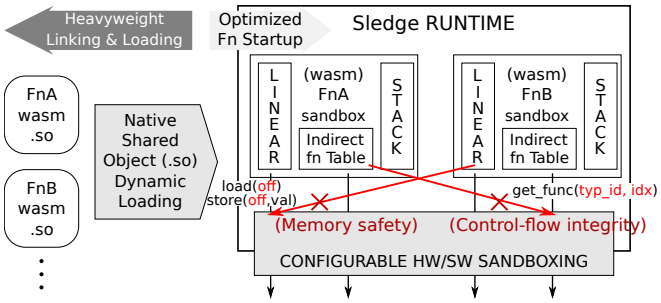

Figure 2. *Sledge* compiler *aWsm* and runtime for memory safety and control-flow integrity of Wasm functions. "Heavyweight Linking and Loading" warms up functions without function sandbox creation/instantation ("Optimized Fn Startup" in the figure).

The *Sledge* compiler, *aWsm*, and a runtime together enable the decoupling of process linking and loading, which is expensive, from the function instantiation, which is optimized for function startup times (Figure 2). *aWsm* compiles code for a function into an elf dynamic library (`.so`) that includes the necessary Wasm safety checks. To load in a new function, the *Sledge* runtime dynamically loads a function's `.so`. When a function execution is triggered, *Sledge* allocates and populates linear memory for the execution of the already linked and loaded Wasm module. By avoiding heavy-weight linking and loading, this approach supports faster function startup time.

Figure 2 helps to demonstrate the ability of *Sledge* compiler and runtime to provide configurable mechanisms for memory and function invocation bounds checks. Software conditional bounds checks for Wasm linear memory represent the most obvious overheads over native code. Such safety checks ("load" and "store" in Figure 2) transform native loads/stores into an addition (from the linear memory base), and a branch (for the bounds check) as shown in Figure 2. We implement these safety checks in the runtime, thus enabling their customization and implementation in C libraries.

We investigate multiple bounds check configurations:

- *No explicit bounds checks* on Wasm *loads* and *stores*. Though this configuration is fastest, it does *not* provide memory safety (thus breaking the sandbox). This is useful for studying the overheads of bounds checking.
- *Software-based bounds checks* that make sure that the *load* and *store* offsets are indeed within the linear memory size of the sandbox. This configuration could limit performance in some systems as it increases the overhead of each *load* and *store*. Note, that LLVM optimizations might lift and remove some redundant checks.
- *Hardware-based bounds checks* using features such as segmentation and MPX [55, 59]. Many processors today have security features that enable pointer validation with limited performance impact on the applications [8, 55, 85].
- *An architecture-based optimization* [33] that elides bounds checks by running the 32-bit Wasm sandbox's linear memory in a separate, aligned 4GB virtual span of address space. Non-overlapping sandboxes ensure memory safety, while virtual memory faults trap illegal accesses.

The *Sledge* runtime also implements software-based safety checks on function pointer invocations to ensure CFI (see "get_func" in Figure 2) by checking that a Wasm module only invokes its accessible functions with valid argument types.

The *Sledge* compiler *aWsm* allows configuring the method for bounds checks at runtime (as shown in Figure 2). This feature enables the runtime to leverage different hardware and software capabilities for bounds check management, while also allowing the edge provider to make performance tradeoffs.

3.3 Single Process Runtime and Resource Management

Existing serverless frameworks for Edge computing, based on VMs and containers, often have a number of special software components spread across separate containers/processes. For example, a "serverless management" component shown in Figures 1 (a)-(c) is essential for request routing and load-balancing across functions in VMs, containers, or processes. However, this approach incurs significant overhead in inter-process communication (IPC) between these components, when crossing protection domains, and therefore, impacting the end-to-end latencies and throughput of executed serverless functions.

The serverless functions are ephemeral and stateless. They often rely on networked storage for persistanse. However, in the existing serverless frameworks, the function isolation boundaries are enforced via strong isolation of VMs and containers (that are inherently heavy-weight). Moreover, the function language runtime and libraries are isolated and replicated as well (see Figures 1 (a)-(c)), thereby increasing the memory footprint of these functions.

The *Sledge* serverless runtime uses **a single process** and leverages Wasm for sandboxing. This design enables a multi-tenant function execution within a single protection domain as shown in Figure 1 (d).

Serverless function resource management. Figure 3 depicts the major components of the *Sledge* runtime for function request processing at the Edge. As it shows, the language runtime and libraries are shared by all the functions executing in the *Sledge* runtime. This approach minimizes

[2] www.github.com/gwsystems/awsm

the memory footprint of each function and enables efficient resource utilization.

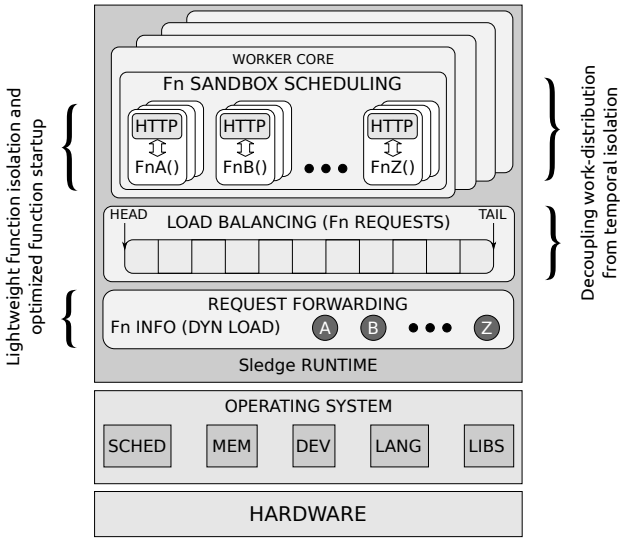

Figure 3. Layers in the *serverless-first, Sledge* framework.

Serverless infrastructure must support *request forwarding* and *function instantiation* to demultiplex the requests. In *Sledge*, this determines a function to be executed in response to a request (based on the connection information and HTTP request). Then the corresponding sandboxes are created for functions execution (shown as FnA() - FnZ() in Figure 3). The *Sledge* runtime decouples the heavy-weight function linking and loading process from *function instantiation* process. This enables the *Sledge* runtime to support the startup latencies of functions at the μ-second level. Each sandbox (with the function request) is then sent to system's cores for the execution. These requests are *load balanced* between the cores. Though existing serverless architectures also include these components, *Sledge* incorporates them into a single UNIX process to remove the additional system overheads.

The APIs from a function to the surrounding environment (including the reply to the client and communication with cloud storage services) are limited to HTTP requests and replies.

3.4 Serverless-First Function Scheduling

The existing serverless frameworks predominantly run functions in VMs and Linux-based containers. While scheduling and load balancing of threads across cores is generally a difficult problem [51], the growing complexity of the Linux kernel [43] can significantly impact every system interaction in a multi-core system [29]. The Linux kernel's scheduling subsystem uses shared runqueues and provides the runqueue synchronization through shared locks. The Inter-Processor Interrupts (IPI) are used to coordinate many aspects in Linux kernel, *e.g.*, scheduling runqueue balancing, work-queues, and TLB coherence. The multi-tenant serverless frameworks suffer from reduced performance due to high context switch overheads and frequent inter-core migrations for enabling temporal isolation.

The serverless executions that are short-lived and event-driven (as described in Section 1) incur significant overheads in the existing systems as the new requests are directly added to the kernel scheduler runqueues and require expensive thread migrations for balancing work. This has motivated a number of systems [12, 40, 63] to bypass the Linux kernel for task scheduling and improved scalability.

To scale and efficiently execute functions, *Sledge* revisits system scheduling to explicitly decouple work distribution from temporal safety (per-function progress). The *Sledge* runtime's design incorporates all functionality in a single process to provide low latencies, while enabling high-churn. In doing so, *Sledge* uses its own facilities for memory isolation, serverless-specific load-balancing, and user-level scheduling of function execution.

The current *Sledge* user-level scheduling is based on:

- preemptive round-robin (RR) scheduling to minimize overheads while also providing the temporal isolation necessary for multi-tenancy;
- a scalable work-stealing deque to shuffle requests between cores, while integrating the dequeueing of new requests into the idle loop of scheduling.

While the user-level scheduling in *Sledge* is much more efficient (by avoiding system calls), it has to address the traditional challenges and issues, such as:

- blocking within the kernel causes that *all* user-level threads are blocked;
- cooperative scheduling is not resilient to the untrusted computations with potentially unbounded execution times.

To address these issues, *Sledge* leverages timer signals on each core as a means to provide preemptive scheduling, integration with an event-driven library for I/O that avoids blocking, and cooperative yields (in the runtime) when the output from system calls is not immediately available.

The *Sledge* runtime is inspired by task-based parallelism frameworks such as OpenMP [61], Cilk [13] and Intel TBB [72] focusing on efficient work distribution of run-to-completion serverless functions or tasks. The *Sledge* runtime also specializes the management of new client requests to avoid head-of-line blocking. When shuffling requests between cores, *Sledge* must enable work-conservation, i.e., a worker core should only go idle if there are no pending requests, and it must scalably balance work. While a global queue is work-conserving, it is not scalable, and while a separate queue per core is scalable, it does not provide work-conservation. Therefore, the *Sledge* runtime leverages a *lock-free, work-stealing deque* [15, 45]) to shuffle requests between cores. Each runtime thread on each core dequeues pending requests to maintain work-conservation, as shown in Figure 3.

Once a function begins execution in a Wasm sandbox, per-core runqueue scheduling is leveraged at the user-level

by providing quantum-based CPU allocation along with preemptive round-robin scheduling. As function instances are user-level sandbox contexts, cooperative switches between them avoid mode changes and hardware protection domain switches. This has a significant impact on the end-to-end function latencies, while enabling a multi-tenant isolation at the Edge.

3.5 Request/Response Serverless Processing

We focus the *Sledge* design around request/response serverless computation[3]. This is motivated by the *Sledge*'s focus on the Edge interaction with IoT and embedded devices, where providing a timely response will best utilize Edge proximity to devices. Matching this motivation, *Sledge* uses HTTP POST requests to pass the arguments (e.g., an image to be classified) to the function for processing. The function's outcome is formatted as the HTTP response. As standards for request/reply processing at the Edge stabilize, this modular mechanism for interacting with a function can be replaced or refined.

The outlined *Sledge* design enables a single node runtime to cater to the execution requirements of serverless at the Edge while optimizing the efficiency and system resource usage of a multi-core system.

The existing C/C++ programs compiled to Wasm leverage POSIX layer without modifications, and *Sledge* currently provides access to this POSIX layer functionality using asynchronous I/O. WebAssembly System Interface (WASI) support is in our roadmap but is out of scope of this paper.

We believe that the multi-server serverless frameworks could be built on top of these fundamental mechanisms to enable higher densities and lower latencies than the existing VM and container-based serverless frameworks.

4 Implementation

Function Management: Figure 4 depicts a single process or address space *Sledge* system, designed to enable multi-tenant serverless functions and efficiently utilize the resource constrained Edge systems. The *Sledge* runtime leverages ahead-of-time (AoT) compiled Wasm modules (as described in Figure 2), loading them at the startup to decouple the heavy-weight function linking and loading from function instantiation. The Wasm modules are loaded using `dlopen` and obtaining the address of the exported *main* function in the module using `dlsym`. Then the runtime creates a TCP socket on a fixed port, defined in a JSON-based configuration file.

A single "listener thread" is pinned to a dedicated core (called a "listener core"). It uses `epoll` to listen to incoming connections on loaded module ports. The "listener core" handles *request forwarding* by listening to incoming requests

[3]Some alternative frameworks (e.g., Google's Firebase [25]) trigger serverless functions execution in response to Cloud storage updates.

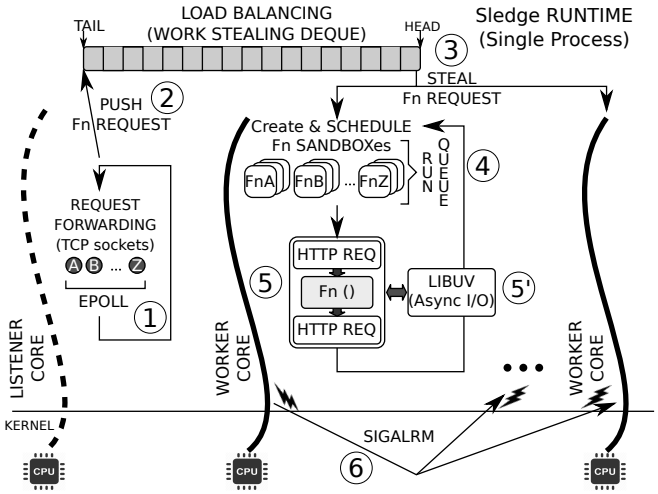

Figure 4. *Sledge* single process serverless runtime. Circled numbers are labels used for explanation of the system functionality flow described and referred in the text.

(labelled (1) in Figure 4) and instantiating function sandboxes for each request to available functions in the runtime. The decoupled function startup enables optimized function sandbox creation, which only involves allocation of required linear memory, a dedicated stack, and a user-level context (tracking *ip*, *sp*, and `mcontext_t`) for user-level scheduling (similar to green threads [32]).

Function Distribution for Load Balancing: The *load balancer* is decoupled from the temporal isolation scheduling and is implemented using a common task queueing data-structure (a global work-stealing deque) to balance sandboxes between "worker cores" [15, 45] as shown in Figure 4. The function sandboxes are pushed on to the *load balancer* for scaling concurrent requests onto a number of cores (labelled (2) in Figure 4). Each "worker core" steals a sandbox from the global work-stealing deque and appends it to the local scheduling runqueue for execution (labelled (3) in Figure 4), thus achieving work-conservation. The data-structure tracking modules, the worker thread `pthread_ts` (which are read-only for the lifetime of the *Sledge* runtime after creation), and the global work-stealing deque are *the only global data-structures* in the runtime. This enables *Sledge* runtime to support predictable latencies for function executions in a multi-core Edge node.

Function Scheduling for Temporal Isolation: To maintain the fairness and temporal isolation required by the multi-tenant function executions, *Sledge* implements preemptive-scheduing on each core, decoupled from *load balancer* for work-distribution. The *Sledge* runtime spawns N "worker threads", each pinned to a separate core and having a core-local *function scheduling* runqueue (with other data-structures implemented using thread-local storage) depicted as (4) in Figure 4. Each "worker thread" unmasks `SIGALRM` (with the

remaining threads keeping it masked by default). This ensures that the software interrupts or signals are always delivered to one of the "worker threads".

Each "worker core" implements a local, preemptive round-robin scheduling for the sandboxes on its run queue, with a time slice set to 5 milliseconds. When the kernel triggers a `SIGALRM`, the "worker core" that receives it, propagates it to the other "worker cores" using `pthread_kill` to enable preemption on multi-core systems ("SIGALRM" or (6) in Figure 4). On each core, preemption causes the `mcontext_t` from the signal handler to be saved to the currently executing sandbox's context, which can only be restored through a signal execution context (as it requires restoring all registers of the sandbox context). A context switch to a preempted thread incurs additional overheads, as it requires a signal execution context, achieved using a `pthread_kill(pthread_self(), SIGUSR1)` to restore all its registers from a previously saved `mcontext_t`. This significantly increases the sandbox context switch costs to be on the orders of the OS thread switch in Linux. Therefore, the time slice in scheduling has strong control over sandboxing preemptions and scheduling overheads, which impact the function's end-to-end latencies.

Function Sandbox Execution: Each function sandbox starts by executing the `main` function inside the Wasm module, reading the HTTP request body as `stdin` (labelled (5) in Figure 4). The `stdout` from the sandbox is then used for HTTP response formation by *Sledge* runtime. Each "worker core" tears down the sandbox memories (stack and linear memory) of each completed sandbox on their local runqueue.

HTTP and Asynchronous Function I/O: Wasm modules, loaded by *Sledge* execute I/O via the system POSIX interface, which includes some system calls that block until completion (*e.g.*, `open, read, write, recv, send`, etc). To enable strong temporal isolation between multi-tenant function executions, the *Sledge* runtime leverages the asynchronous, event-driven I/O using *libuv* [47] ("libuv (Async I/O)" or (5') in Figure 4). libuv is a multi-platform support library with a focus on asynchronous I/O based on event loops (`uv_loop_t`). It was primarily designed for use in Node.js but is also used by many other projects.

Each "worker core" has a core-local libuv event loop (in its thread-local storage). The functions (executing in that thread) queue the I/O requests in the core-local event loop and use cooperative sandbox scheduling to block on I/O. The *function scheduling* in each "worker core" checks for pending I/O before scheduling the function sandboxes from the runqueue, enabling the scheduler to control the number of events to process. Event-driven I/O is provided by registered libuv callbacks, which wake the sandboxes up and make them runnable by appending them to the core-local runqueue.

5 Evaluation

First, we evaluate our *Sledge* compiler *aWsm* and runtime on x86_64 and AArch64 architectures using two systems:

- Dell Precision 7820 workstation with 16 cores Intel Xeon Silver 4216 2.1 GHz processor and 16 GB memory;
- Raspberry Pi 4 Model B with Quad-core Cortex-A72 (ARM v8) 64-bit SoC at 1.5GHz having 4 GB physical memory.

We use a Ubuntu 18.04 64bit OS, and `clang / LLVM` compiler version 8.0.

Then by using various Edge workloads, we analyze *Sledge* serverless runtime performance on x86_64 and compare it to *Nuclio* [57], the open-source serverless framework.

5.1 WebAssembly Performance

By providing a mechanism for lightweight isolation, the Wasm sandbox enables *Sledge* to run functions with high churn and low latency. This is paired with the memory-safety and control-flow integrity properties discussed in §3.1.

Given the centrality of Wasm to promise near-native execution performance while enabling strong, yet, lightweight isolation, we designed an experiment to evaluate the execution times of PolyBench/C 4.2.1 applications (as in [33]) on *Sledge* using our *aWsm* compiler and several popular Wasm runtimes.

The runtimes that we evaluate are:

- **Wasmer** v0.12.0 [78], a just-in-time (JIT) compilation runtime that uses Cranelift [19] as its default compiler (LLVM and single-pass are supported) and allows caching of JIT compiled objects to improve performance of further executions. We evaluate performance for Cranelift based compilation with enabled caching of JIT compiled objects.
- **WAVM** v0.0.0-prerelease [79], a JIT compilation runtime that compiles Wasm objects using the LLVM compiler with the caching of JIT compiled objects enabled.
- **Node.js** v12.9.1 [56], which executes Wasm objects within the V8 runtime. **Emscripten** v1.39.5 [22] compiler (LLVM-based) is used to generate JavaScript wrappers.
- **Lucet** v0.4.1 [4] native Wasm compiler and runtime. The `lucetc` ahead-of-time (AoT) compiler leverages Cranelift for compilation of Wasm objects and generates native shared objects. The `lucet-wasi` runtime loads a single shared object and executes it in a sandboxed environment.
- ***Sledge+aWsm***, a native, AoT-generated executable created by the aWsm compiler and dynamically loaded as a shared object and executed as a single sandbox in the *Sledge* runtime. The aWsm compiler generates LLVM IR from Wasm bytecode and outputs a native shared object as discussed in §3.2. This runtime is configured to use virtual memory based bounds management (isolating 32-bit Wasm sandboxes into 4GiB regions) and we refer to this simply as *Sledge+aWsm* in Figure 5.
- ***Sledge+aWsm*-mpx**, a native executable generated by the aWsm compiler with a *Sledge* runtime configured

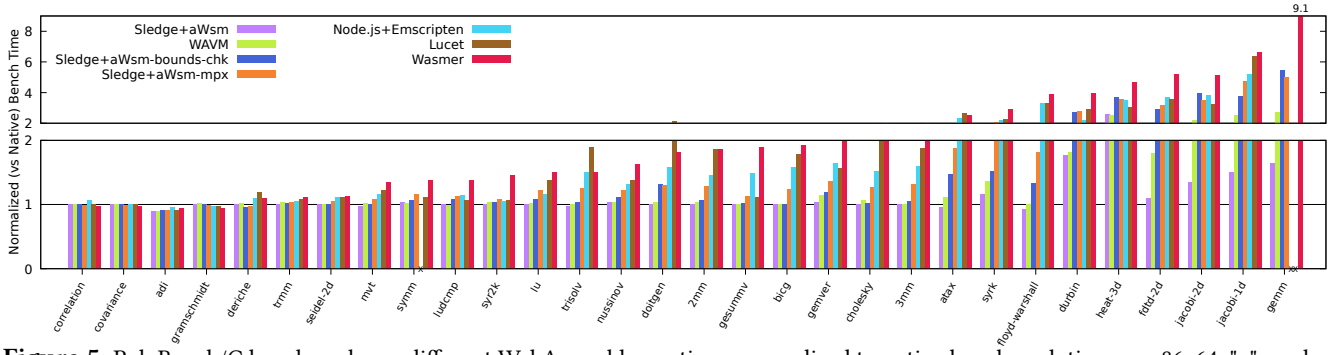

Figure 5. PolyBench/C benchmarks on different WebAssembly runtimes normalized to native benchmark time on x86_64. "x" mark on x-axis indicates failure to execute benchmark on a runtime. Node.js+Emscripten failed for `gemm` and `symm`. Lucet failed for `gemm`.

	x86_64							*Raspberry Pi (ARM v8)*	
Wasm Runtime	Wasmer	WAVM	Node.js+ Emscripten	Lucet	*Sledge+aWsm-* bounds-chk	*Sledge+* mpx	*Sledge+ aWsm*	*Sledge+aWsm-* bounds-chk-rpi	*Sledge+aWsm-* rpi
Slowdown (AM)	149.8%	28.1%	84.0%	92.8%	62.7%	75.1%	13.4%	36.7%	6.74%
Slowdown (GM)	101.6%	20.5%	62.3%	68.9%	38.4%	51.6%	9.9%	26.86%	5.0%
SD	194.09	53.09	107.84	117.25	116.14	113.41	34.65	60.3	19.38

Table 1. Arithmetic mean (Slowdown (AM)) and geometric mean (Slowdown (GM)) of % slowdowns, and the standard deviations (SD) for arithmetic mean in different Wasm runtimes. The x86_64 results summarize the results from Figure 5. For brevity, we only include summarized results for AArch64 on Raspberry Pi in the right-most columns of the table (systems with -rpi suffix).

with Intel MPX [55, 59] hardware-based bounds checks for memory safety.

- ***Sledge+aWsm*-bounds-chk**, a native executable generated by the aWsm compiler with a *Sledge* runtime configured with naive software-based bounds checks for memory safety discussed in §3.2.
- ***Sledge+aWsm*-rpi**, a native executable generated by the aWsm compiler for AArch64 on Raspberry Pi with a *Sledge* runtime configured with virtual memory based bounds management similar to *Sledge*.
- ***Sledge+aWsm*-bounds-chk-rpi**, a native executable generated by the aWsm compiler for for AArch64 on Raspberry Pi with a *Sledge* runtime configured with software-based bounds checks similar to *Sledge-bounds-chk*.
- **Native** executable compiled with `clang -O3` for baseline.

Figure 5 presents the *normalized to native* execution time of PolyBench/C benchmarks, averaged over 15 iterations for different Wasm runtimes on x86_64 system. To match the methodology used in [33], we used 15 iterations in this experiment. Table 1 (left side, that represents x86_64 results) shows the arithmetic and geometric means of PolyBench/C benchmarks as a percentage slowdown compared to native and the standard deviations for different Wasm runtimes of different benchmarks on x86_64 system.

Discussion. As shown in Figure 5, the Cranelift-based runtimes show a significant slowdown compared to native (149.8% in Wasmer and 92.8% in Lucet), reflecting Cranelift's emphasis on fast compilation times [6]. WAVM, a JIT compiler runtime that leverages LLVM for code generation, shows a relatively smaller 28.1% slowdown. Node.js+Emscripten performance show the sub 10% slowdown for a subset of 7 applications (as also discussed in [33]), which however grows to 84% across all applications.

The *aWsm* compiler and *Sledge* runtime is 13.4% slower (on average) than native and at least 14.7% faster than the fastest Wasm runtimes (WAVM) as shown in Slowdown (AM) of Table 1. The naive software-based bounds checks in *Sledge+aWsm*-bounds-chk require linear memory offset and bounds checks (which LLVM might optimize), which adds overhead to each *load* and *store*, in the worst case. This generates additional slowdown over that of *Sledge+aWsm*, which uses virtual memory-based bounds management to maintain memory isolation and avoids bounds check overheads.

Sledge+aWsm-mpx with Intel MPX hardware-based support is counter-intuitively 12.4% slower than *Sledge+aWsm-bounds-chk* with software-based bounds checks, and it is 61.7% slower than *Sledge*, reflecting the expensive operations [59] required in storing and loading of the bounds.

We also evaluated the *Sledge+aWsm* performance with static compilation without any bounds checks mechanism to compare the overheads and we observed that the arithmetic mean was 0.3% faster than *Sledge+aWsm*. The geometric slowdown (Slowdown (GM) in Table 1) also confirms that *Sledge+aWsm* performs better than other LLVM-based and significantly better than the Cranelift-based runtimes. These results show that the Wasm sandboxing in *Sledge+aWsm* is efficient and offers a good foundation for isolation in a serverless runtime.

***Sledge+aWsm* on Raspberry Pi.** Lately, the ARM v8 processors have gained significant traction in the Cloud and Edge computing worlds for their performance characteristics [10, 64]. The Raspberry Pi devices, based on ARM v8, are

powerful enough to support Edge computing, with attractive form-factor and cost. We ported our *aWsm* compiler and the *Sledge* serverless runtime to AArch64 to evaluate their performance for ARM-based Edge systems. Table 1 (right part) presents the arithmetic means of % slowdown compared to native code execution and their standard deviation for *Sledge+aWsm* running native executables on AArch64.

The *Sledge* runtime with software-based bounds checks (*Sledge+aWsm-bounds-chk-rpi*) is 36.7% slower than native, as it incurs software overheads in bounds checks similar to (*Sledge+aWsm-bounds-chk*) on x86_64. More importantly, the *Sledge+aWsm-rpi* with virtual memory based sandboxing shows only 6.7% slowdown compared to native on ARM AArch64, thus, demonstrating that Wasm performance on ARM v8 processor is close to native.

Memory footprint. The memory footprint of functions has a significant impact on "cold-start" performance and scalability at the resource-constrained Edge. The single-process *Sledge* runtime binary size is **359 KB**. Moreover, it enables functions to share the library dependencies, while providing a strong spatial and temporal isolation for multi-tenant functions executions. The AoT compiled shared object sizes are between **108 KB-112 KB**, which is significantly smaller than VM- and container-based function isolation (often in 10s to 100s of MBs [18, 68]).

5.2 Serverless Function Performance

Nuclio [57] is a serverless framework (shown in Figure 1 (c)) that provides more performant low-latency processing by using containers for multi-tenant isolation, while concurrently executing functions in separate processes within those containers. *Nuclio* allows serverless functions to be invoked directly from a client. Therefore, *Nuclio* can be deployed as a single node without load balancers or function-request-forwarding subsystems. These characteristics make *Nuclio* one of the most attractive open-source serverless solutions for the Edge [46]. Given that *Nuclio* is one of the fastest existing serverless solutions [46], we evaluate and compare the latency and throughput of serverless functions executed on *Sledge* and *Nuclio* platforms. The *Nuclio* framework does not yet support Raspberry Pi, and therefore, we perform our experiments only on x86_64 processor.

***Nuclio* Setup** We deployed *Nuclio* (version 1.3.3) as a single node without any of the optional components to support larger, distributed, multi-node, serverless deployments. We use the shell based function processor (version 0.7.1) to run native binaries on the *Nuclio* serverless framework. The function processor containers in *Nuclio*, when deployed, are 96.4MBs in size, which includes the *Nuclio* shell runtime, HTTP event listener and the native executable of the function in each experiment. The *Nuclio* function processor is configured with a maxWorker value, which determines the maximum number of concurrent worker processes that can be spawned. Given the varied performance characteristics of different benchmarks (I/O intensive vs CPU intensive), we tuned this to optimize the throughput of a basic ping function, which performs no computation and only replies with a single byte. We observed the optimal throughput with a worker count between 16 and 19, and therefore use 16 workers to match the 16 cores in our server machine.

***Sledge* Setup** The *Sledge* serverless runtime uses virtual memory based bounds management for function sandbox memory isolation (similar to "*Sledge+aWsm*" in Figure 5). The runtime has 15 worker cores for function instantiation and scheduling, and a listener core for request forwarding and load balancing (as discussed in Section 4 and Figure 4).

The client and server machines both use a 10Gtek 10GbE PCI-E X8 Network Card X540-10G-2T connected via a NETGEAR 8-Port 10G Ethernet Smart Managed Plus Switch (XS708E). On the client-side, we use Apache Bench (version 2.3) in all serverless experiments. We observed that the minimum average round-trip network latency exhibited by different serverless experiments regardless of the workload is 0.122*ms*. We measure this by running a simple web server on the server machine and measured the latency for 10k requests using `ab` on the client machine. In all the experiments below, we have measured the throughput in Kbps for different executions in *Sledge* and *Nuclio*, and verified that the network bandwidth is not a bottleneck.

Varying concurrency. We first evaluate the latency and throughput properties of *Sledge* with a simple ping functionality, with increasing concurrency. Figure 6 shows the throughput and average latencies for *Sledge* and *Nuclio*.

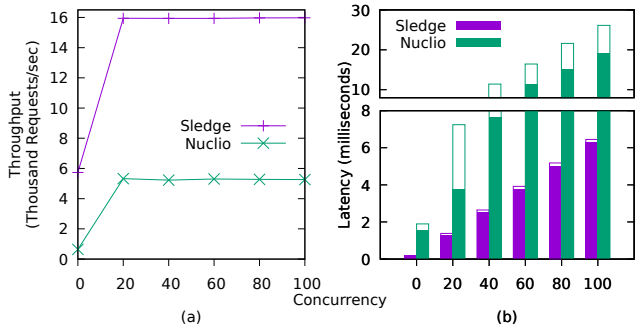

Figure 6. (a) Throughput and (b) Latency of a Ping serverless function with varying concurrency. (Solid and empty bars indicate average and 99%-tile latencies respectively, for 10k iterations.)

Discussion. The *Sledge* serverless runtime shows consistently high throughput – about 3 times *Nuclio* – across varied concurrency levels in Figure 6(a) and significantly lower average and 99%-tile latencies compared to *Nuclio* in Figure 6(b). The multi-process *Nuclio* runtime, executing native function binaries, incurs IPC, context switch, and scheduling overheads of traditional system scheduling, in addition to the overheads in function instantiation (using `fork` + `exec`). In contrast, our serverless-first *Sledge* framework that leverages light-weight Wasm sandboxing and decouples (1) function

linking and loading from function instantiation, and (2) work-distribution from user-level sandboxing scheduling for temporal isolation, avoids many of these overheads (in spite of the Wasm slowdown compared to native). *Sledge* demonstrates promising ultra-low latencies and high throughputs.

Varying payload sizes. Edge functions often have strict latency and data processing requirements. It is necessary for the serverless runtimes to provide high throughput, while enabling low end-to-end latencies with varying payload sizes for data processing. Next, we evaluate the latency and throughput of *Sledge* with a network-intensive workload.

Figure 7 presents the throughput and average latencies for *Sledge* and *Nuclio* systems for network I/O intensive function. This function receives a payload (from 1KB to 1MB), which it copies into a buffer, and then writes back out to `stdout`, which is sent as the HTTP response body.

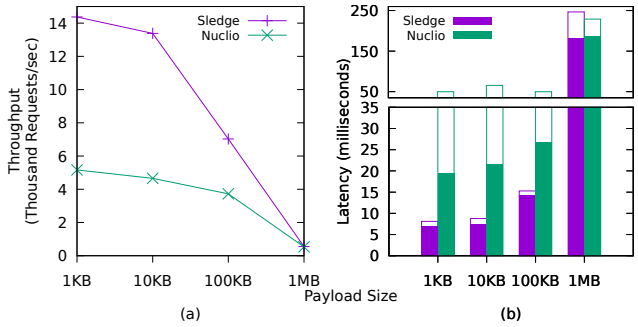

Figure 7. (a) Throughput and (b) Latency of a network-transfer serverless function with 100 concurrent conn. (Solid bars indicate avg. latencies at 10k iterations, empty bars indicate p99 latencies.)

Discussion. The *Sledge* runtime has approximately 2.8 times higher throughput than *Nuclio* as shown in Figure 7(a) and approximately 2.8 times lower latencies in *Sledge* (Figure 7(b)) for 1KB and 10KB request and response sizes. With larger transfer sizes, the costs of copying data (for example, through POSIX `read` and `write` APIs) dominates the execution, and *Sledge* performance approaches that of *Nuclio*. We believe that given a limited bandwidth and power of IoT and smart devices, utilizing the serverless Edge for real-time functionality and data processing, will mostly use the *small request/response sizes*.

We additionally run experiments (not shown) with CPU-bound functions of various computation times. As functions become increasingly CPU-bound, the performance of *Sledge* gets closer to *Nuclio*. Though many functions may be CPU-bound, we believe that a typical use case for the real-time Edge will be light-weight functions aiming at fast responses.

Application study. Next, we study the performance of real-world Edge applications that perform a mix of network transfers and computations. Since IoT and smart devices are often resource constrained, they might rely on Edge systems to support compute-intensive functions from Neural Network libraries, image recognition and manipulation, and other object detection functionality.

The applications we evaluate are:

- *Arm's CMSIS-NN [17] for CIFAR10 [44]*: Reads a 1.9 KB 32x32 PNG image of an airplane from `stdin`, executes a classifier of 10 classes (airplanes, cars, birds, cats, deer, dogs, frogs, horses, ships, and trucks), and writes the number associated with the resulting class to `stdout`.
- *GOCR [30]*: An Optical Character Recognition (OCR) application that reads a portable-bitmap format (PBM) image from `stdin` and writes the ASCII text recognized from the image to `stdout`.
- *SOD [71] RESIZE*: Reads a 28.9 KB flower JPEG from `stdin`, resizes it by half, and outputs as PNG to `stdout`.
- *SOD [71] License Plate Detection (LPD)*: Reads a 96.6 KB JPEG containing a license plate from `stdin`, generates a bounding box around the license plate, and outputs a 96.6 KB JPEG image with a box drawn.
- *GPS with TinyEKF [74]*: Reads Global Positioning System (GPS) state and input position matrices from `stdin`, uses TinyEKF, a simple Extended Kalman Filter [66], to predict a future position, which it writes to `stdout`. Given EKF's need for a state from the previous executions, it returns to the client that state, and relies on it to pass it along with each request.

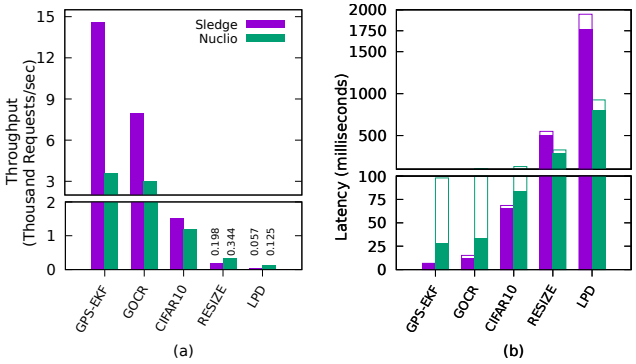

Figure 8. (a) Throughput and (b) Latency of real-world serverless functions at 100 concurrent connections. (Solid bars indicate avg. latencies for 10k iterations, empty bars indicate 99%-tile latencies.)

	Native		*Sledge*			
	Avg	99%	Avg	(norm)	99%	(norm)
GPS-EKF	27μs	32μs	30μs	(1.09)	33μs	(1.03)
GOCR	623μs	704μs	924μs	(1.48)	1015μs	(1.44)
CIFAR10	5.54ms	6.46ms	8.31ms	(1.49)	9.27ms	(1.43)
RESIZE	36.1ms	37.4ms	53ms	(1.46)	55ms	(1.46)
LPD	96.3ms	99.3ms	176.3ms	(1.83)	180ms	(1.81)

Table 2. Execution time of real-world serverless functions in *Sledge* and native code execution. (Averaged over 1k iterations).

Figure 8 presents throughput and average latencies in *Sledge* and *Nuclio* systems processing real-world applications. To provide an insight into Wasm execution performance and possible overhead, Table 2 provides the runtimes of the same functions, executed outside of serverless framework.

These demonstrate the Wasm versus native performance disparities.

Discussion. With the exception of GPS-EKF, the executed applications are approx. 45% slower than native code due to Wasm overhead, as shown in Table 2. In spite of this, the *serverless-first* approach (redesigned from the ground up) enables *Sledge* runtime to provide high throughput and low latencies for all applications (except those that are significantly computation-bound: RESIZE and LPD). The throughput of GPS-EKF application in *Sledge* is **4 times** higher than *Nuclio* in Figure 8(a) and its average latency in *Sledge* is **4 times** faster than *Nuclio* in Figure 8(b). Similarly, the throughput of GOCR application in *Sledge* is 2.9 times higher than *Nuclio* and the average latency in *Sledge* is 2.9 times faster than *Nuclio* and for CIFAR10, the throughput is 1.36 times higher and the average latency is 1.36 times faster than *Nuclio*.

This demonstrates the efficiency of the *Sledge* design, which is especially pronounced and strongly manifests itself for less heavy computational functions. It also shows that further work on optimizing Wasm compilers will be beneficial. The *Sledge* runtime is optimized for short computations at the Edge. It might under-perform for CPU intensive functions (due to Wasm-related overheads compared to their native execution). Wasm-based executions of RESIZE is 46% slower and LPD is 81% slower than native as shown in Table 2. This explains the lower throughput and higher latencies of these applications in *Sledge* runtime compared to their processing in *Nuclio*.

Function creation latencies and churn. Driven by data processing from IoT and smart devices, the Edge computing systems will require to handle high request rates due to reduced network latency between mobile IoT devices and the servers at the Edge (*e.g.*, 5G stations). The existing serverless frameworks, based on VMs and containers, exhibit high "cold-start" latencies (discussed in §2), thus hindering the high churn required by these devices. The *Nuclio* serverless framework enables the function processors (containers) to be persistent. The serverless management within the container forks a process for each invocation, thus incurring only the cold-start latency of process creation. To evaluate the *Sledge* runtime's ability to support a high churn of requests and to compare it against *Nuclio* serverless framework, we measure the latency of the GPS-EKF application, using `fork + exec` on native system and the sandbox execution in *Sledge*.

	99%	Avg.
`fork + exec + wait`	588μs	487μs
Sledge Sandbox	146μs	61μs

Table 3. Churn benchmarks (μs) for GPS-EKF in a *Sledge Sandbox* and using `fork + exec + wait` on native. (Averaged over 10k iterations)

Table 3 shows the latency of process creation in a *Sledge* standalone sandbox and `fork+exec+wait` in native system.

Discussion. The optimized function instantiation in *Sledge* is significantly faster than native `fork + exec`, because it does not need to track and replicate the kernel state such as files and signals, and it avoids overheads of creating and immediately overwriting the newly created address space (e.g., `fork` then `exec`). More importantly, despite copy-on-write, the costs of `fork` and `exec` depend on the size of the running process and executable being `exec`ed, which can be significantly higher for large runtimes and functions. Therefore, decoupling the function linking and loading from the function instantiation in *Sledge* enables significantly lighter-weight (30μs) function startup times, and efficiently managing a high churn of request rates in the Edge systems.

6 Conclusion

The current surge of novel applications for IoT, autonomous vehicles, and pervasive surveillance motivates the need of time-critical, high bandwidth processing of sensor data at the Edge. To maintain low-latencies while scaling multi-tenancy within the Edge restricted resources, this paper presents a *serverless-first* runtime designed to optimize for the *short-lived* and *event-driven* properties of serverless functions. The *Sledge* compiler and runtime enable light-weight function isolation by leveraging the sandboxing properties of Wasm.

Results show that *serverless-first Sledge* runtime enables low-latency serverless execution, while efficiently managing concurrency and work distribution. For the most latency-sensitive Edge applications with fast runtimes, *Sledge* has latencies **4x** lower than *Nuclio* (a high-performance, container-based serverless infrastructure), with a **4x** higher throughput. We also show that *Sledge*'s compiler *aWsm* generates sandboxed code that executes within 1.1 times compared to native code for 24 out of 30 PolyBench/C benchmarks. We believe this demonstrates that a serverless runtime, optimized by leveraging light-weight Wasm-based isolation and scheduling bypass of traditional kernel scheduling, holds a significant promise for strict requirements of future Edge architectures. The proposed framework opens up a set of interesting opportunities for customized performance management of users' serverless functions, which we plan to investigate in our future work.

7 Acknowledgements

We would like to thank the anonymous reviewers and our shepherd Christof Fetzer for a valuable feedback and help in improving the paper presentation. This work was partially completed during P. K. Gadepalli's summer internship in 2019 at Arm Research. We would like to thank the support from the NSF through awards CNS-1815690 and CPS-1837382, and from Arm and SRC through Task 2911.001. The views of this paper does not necessarily reflect those of NSF nor SRC.

References

[1] 2018. Choosing the right amount of memory for your AWS Lambda Function: https://blog.symphonia.io/the-occasional-chaos-of-aws-lambda-runtime-performance-880773620a7e .

[2] 2018. The Occasional Chaos of AWS Lambda Runtime Performance: https://medium.com/@raupach/choosing-the-right-amount-of-memory-for-your-aws-lambda-function-99615ddf75dd.

[3] 2019. Announcing Lucet: Fastly's native WebAssembly compiler and runtime. https://www.fastly.com/blog/announcing-lucet-fastly-native-webassembly-compiler-runtime

[4] 2019. Lucet: https://github.com/fastly/lucet.

[5] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2005. Control-Flow Integrity. In *Proc. of the 12th ACM Conference on Computer and Communications Security (CCS)*.

[6] Syrus Akbary. 2019. A WebAssembly Compiler Tale, https://medium.com/wasmer/a-webassembly-compiler-tale-9ef37aa3b537.

[7] Alibaba Cloud Functions 2019. Alibaba Function Compute: https://www.alibabacloud.com/products/function-compute.

[8] Arm 2019. Armv8.5-A Memory Tagging Extension, https://semiengineering.com/a-memory-tagging-extension/.

[9] Gabriel Aumala, Edwin F. Boza, Luis Ortiz-Avilés, Gustavo Totoy, and Cristina Abad. [n.d.]. Beyond Load Balancing: Package-Aware Scheduling for Serverless Platforms. In *19th IEEE/ACM Intl. Symposium on Cluster, Cloud and Grid Computing, CCGRID 2019.*

[10] AWS Goes All In On Arm-Based Graviton2 Processors 2019. AWS Goes All In On Arm-Based Graviton2 Processors With EC2 6th Gen Instances, https://www.forbes.com/sites/moorinsights/2019/12/03/aws-goes-all-in-on-arm-based-graviton2-processors-with-ec2-6th-gen-instances/.

[11] AWS Lambda 2019. AWS Lambda: https://aws.amazon.com/lambda/.

[12] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. 2014. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI)*.

[13] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. 1995. Cilk: An Efficient Multithreaded Runtime System. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 207–216.

[14] Sol Boucher, Anuj Kalia, David G. Andersen, and Michael Kaminsky. 2018. Putting the "Micro" Back in Microservice. In *Proc. of the 2018 Usenix Annual Technical Conference* (Boston, MA, USA) *(USENIX ATC '18)*.

[15] David Chase and Yossi Lev. 2005. Dynamic Circular Work-Stealing Deque. In *SPAA '05: Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*.

[16] Y. Chen, Q. Feng, and W. Shi. 2018. An Industrial Robot System Based on Edge Computing: An Early Experience. In *USENIX Workshop on Hot Topics in Edge Computing (HotEdge 18)*.

[17] CMSIS NN 2019. CMSIS NN Software Library, https://arm-software.github.io/CMSIS_5/NN/html/index.html.

[18] Containers vs Virtual Machines 2019. What is a Container?: A standardized unit of software, https://www.docker.com/resources/what-container.

[19] Cranelift 2019. Cranelift Code Generator, https://cranelift.readthedocs.io/en/latest/.

[20] Docker 2018. Docker: https://www.docker.com/.

[21] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer. 2003. Xen and the Art of Virtualization. In *Proc. of the ACM Symposium on Operating Systems Principles (SOSP)*.

[22] Emscripten 2019. Emscripten: https://emscripten.org/index.html.

[23] fastly-serverless 2019. Fastly Expands Serverless Capabilities With the Launch of Compute@Edge, https://www.fastly.com/press/press-releases/fastly-expands-serverless-capabilities-launch-compute-edge.

[24] Henrique Fingler, Amogh Akshintala, and Christopher J. Rossbach. 2019. USETL: Unikernels for Serverless Extract Transform and Load Why Should You Settle for Less? *(APSys'19)*.

[25] Firebase 2019. Firebase helps mobile and web app teams succeed. https://firebase.google.com/.

[26] Firecracker 2019. Firecracker: https://firecracker-microvm.github.io/.

[27] Firecracker-Barr [n.d.]. Jeff Barr: "Firecracker - Lightweight Virtualization for Serverless Computing." https://aws.amazon.com/blogs/aws/firecracker-lightweight-virtualization-for-serverless-computing, November 2018.

[28] P. K. Gadepalli, G. Peach, L. Cherkasova, R. Aitken, and G. Parmer. 2019. Challenges and Opportunities for Efficient Serverless Computing at the Edge. In *37th IEEE Symp. on Reliable Distributed Systems (SRDS)*.

[29] Phani Kishore Gadepalli, Gregor Peach, Gabriel Parmer, Joseph Espy, and Zach Day. 2019. Chaos: a System for Criticality-Aware, Multi-core Coordination. In *25th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*.

[30] GOCR 2019. GOCR: open-source character recognition, http://jocr.sourceforge.net/index.html.

[31] Google Cloud Functions 2019. Google Cloud Functions: https://cloud.google.com/functions/.

[32] Green Threads 2019. Many-to-One / Green Threads: https://docs.oracle.com/cd/E19455-01/806-3461/6jck06gqe/.

[33] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the Web Up to Speed with WebAssembly. In *Proc. of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '17)*.

[34] Adam Hall and Umakishore Ramachandran. 2019. An Execution Model for Serverless Functions at the Edge. In *Proceedings of the International Conference on Internet of Things Design and Implementation (IoTDI '19)*.

[35] Adam Hall and Umakishore Ramachandran. 2019. An Execution Model for Serverless Functions at the Edge. In *IoTDI*.

[36] 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)*.

[37] IBM Cloud Functions 2019. IBM Cloud Functions: https://cloud.ibm.com/functions.

[38] Abhinav Jangda, Bobby Powers, Emery D. Berger, and Arjun Guha. 2019. Not So Fast: Analyzing the Performance of WebAssembly vs. Native Code. In *USENIX Annual Technical Conference (ATC 19)*.

[39] 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. (2019). http://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-3.html

[40] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. 2019. Shinjuku: Preemptive Scheduling for μ second-scale Tail Latency. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*.

[41] Kata Container 2019. Kata Containers: https://katacontainers.io/.

[42] Jeongchul Kim and Kuyngyong Lee. 2019. FunctionBench: A Suite of Workloads for Serverless Cloud Function Service. In *Proceedings of the IEEE International Conference on Cloud Computing*.

[43] Ricardo Koller and Dan Williams. 2017. Will Serverless End the Dominance of Linux in the Cloud?. In *Proc. of the 16th Workshop on Hot Topics in Operating Systems (HotOS '17)*.

[44] Alex Krizhevsky. 2010. Convolutional deep belief networks on cifar-10, https://www.cs.toronto.edu/ kriz/conv-cifar10-aug2010.pdf.

[45] Nhat Minh Lê, Antoniu Pop, Albert Cohen, and Francesco Zappa Nardelli. 2013. Correct and Efficient Work-stealing for Weak Memory Models. In *Principles and Practice of Parallel Programming (PPoPP '13)*.

[46] Junfeng Li, Sameer G. Kulkarni, K. K. Ramakrishnan, and Dan Li. 2019. Understanding Open Source Serverless Platforms: Design Considerations and Performance. In *Proc. of the 5th Intl. Workshop on Serverless Computing (WOSC '19)*.

[47] libuv 2012. libuv: Asynchronous I/O made simple, http://libuv.org/.

[48] Linux Containers 2019. Linux Containers: https://linuxcontainers.org/.

[49] Wes Lloyd, Shruti Ramesh, Swetha Chinthalapati, Lan Ly, and Shrideep Pallickara. 2018. Serverless Computing: An Investigation of Factors Influencing Microservice Performance. In *IEEE Intl. Conference on Cloud Engineering (IC2E)*.

[50] LLVM 2019. The LLVM Compiler Infrastructure, https://llvm.org/.

[51] Jean-Pierre Lozi, Baptiste Lepers, Justin Funston, Fabien Gaud, Vivien Quéma, and Alexandra Fedorova. 2016. The Linux Scheduler: A Decade of Wasted Cores. In *Proceedings of the Eleventh European Conference on Computer Systems (Eurosys '16)*.

[52] 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)*.

[53] Microsoft Azure Functions 2019. Microsoft Azure Functions: https://azure.microsoft.com/en-us/services/functions/.

[54] Anup Mohan, Harshad Sane, Kshitij Doshi, Saikrishna Edupuganti, Naren Nayak, and Vadim Sukhomlinov. 2019. Agile Cold Starts for Scalable Serverless. In *Proc. of the 11th USENIX Conference on Hot Topics in Cloud Computing*.

[55] MPX 2013. Introduction to Intel Memory Protection Extensions, https://software.intel.com/en-us/articles/introduction-to-intel-memory-protection-extensions.

[56] Node.js 2019. Node.js: Node.js is a JavaScript runtime built on Chrome's V8 JavaScript engine. https://nodejs.org/en/.

[57] nuclio 2019. Nuclio: Automate the Data Science Pipeline with Serverless Functions, https://nuclio.io.

[58] 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)*.

[59] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. 2018. Intel MPX Explained: A Cross-layer Analysis of the Intel MPX System Stack. *Proc. of the ACM on Measurement and Analysis of Computing Systems* (2018).

[60] OpenFaaS 2019. OpenFaaS: Serverless Functions, Made Simple. https://openfaas.com.

[61] openmp [n.d.]. OpenMP: http://www.openmp.org, retrieved 9/21/12.

[62] PolyBench/C 2019. PolyBench/C: the Polyhedral Benchmark suite, https://web.cse.ohio-state.edu/ pouchet.2/software/polybench/.

[63] George Prekas, Marios Kogias, and Edouard Bugnion. 2017. ZygOS: Achieving Low Tail Latency for Microsecond-Scale Networked Tasks. In *Proc. of the 26th Symposium on Operating Systems Principles*.

[64] Qualcomm New Model for Computing Built on Arm 2017. Qualcomm is Bringing an Exciting New Model for Computing, and It's Built on Arm, https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/qualcomm-bringing-exciting-new-model-for-computing-built-on-arm.

[65] 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 *Proc. of the Workshop on Hot Topics in Operating Systems (HotOS '19)*.

[66] Maria Isabel Ribeiro. 2004. Kalman and extended kalman filters: Concept, derivation and properties. (2004).

[67] serverless [n.d.]. State of the Cloud Report 2019, http://www.rightscale.com/lp/state-of-the-cloud.

[68] Serverless 2019. Virtual Machines vs Containers vs Serverless Computing: Everything You Need to Know, https://dotcms.com/blog/post/virtual-machines-vs-containers-vs-serverless-computing-everything-you-need-to-know.

[69] Simon Shillaker and Peter Pietzuch. 2020. Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing.

[70] Snoyman 2019. Michael Snoyman: "Serverless Rust using WASM and Cloudflare", https://tech.fpcomplete.com/blog/serverless-rust-wasm-cloudflare.

[71] SOD 2019. SOD - An Embedded Computer Vision and Machine Learning Library, https://sod.pixlab.io/index.html.

[72] tbb [n.d.]. Intel Thread Building Blocks: http://threadingbuildingblocks.org/, retrieved 9/21/12.

[73] Jörg Thalheim, Pramod Bhatotia, Pedro Fonseca, and Baris Kasikci. 2018. Cntr: Lightweight OS Containers. In *USENIX Annual Technical Conference (USENIX ATC*.

[74] TinyEKF 2019. TinyEKF: Lightweight C/C++ Extended Kalman Filter with Python for prototyping, https://github.com/simondlevy/TinyEKF.git.

[75] Varda October 1, 2018. Kenton Varda: "WebAssembly on Cloudflare Workers", https://blog.cloudflare.com/webassembly-on-cloudflare-workers/.

[76] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. 1993. Efficient Software-Based Fault Isolation. In *Proc. of the 14th ACM Symposium on Operating Systems Principles (SOSP)*.

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

[78] Wasmer 2019. Wasmer: Run any code on any client. With WebAssembly and Wasmer. https://wasmer.io/.

[79] WAVM 2019. WAVM: WAVM is a WebAssembly virtual machine, designed for use in non-web applications. https://wavm.github.io/.

[80] WebAssembly Security 2020. WebAssembly Security, https://webassembly.org/docs/security/.

[81] White Paper 2017. White Paper of Edge Computing Consortium, https://www.iotaustralia.org.au/wp-content/uploads/2017/01/White-Paper-of-Edge-Computing-Consortium.pdf.

[82] White Paper 2019. Security Overview of AWS Lambda. https://d1.awsstatic.com/whitepapers/Overview-AWS-Lambda-Security.pdf

[83] Windows Containers 2019. Windows Containers: https://docs.microsoft.com/en-us/virtualization/windowscontainers/about/.

[84] William Wong. 2017. VM, Containers, and Serverless Programming for Embedded Developers. (2017). https://www.electronicdesign.com/embedded-revolution/vm-containers-and-serverless-programming-embedded-developers

[85] Jonathan Woodruff, Robert N.M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. 2014. The CHERI Capability Model: Revisiting RISC in an Age of Risk. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*.

[86] Cui Yan. 2017. How does language, memory and package size affect cold starts of AWS Lambda? 2017, https://read.acloud.guru/does-coding-language-memory-orpackage-size-affect-cold-starts-of-aws-lambda-a15e26d12c76.

[87] Tian Zhang, Dong Xie, Feifei Li, and Ryan Stutsman. 2019. Narrowing the Gap Between Serverless and Its State with Storage Functions. In *Proc. of the ACM Symposium on Cloud Computing* (Santa Cruz, CA, USA) *(SoCC '19)*.