FAASM: Lightweight Isolation for Efficient Stateful Serverless Computing

Simon Shillaker and Peter Pietzuch, *Imperial College London*

https://www.usenix.org/conference/atc20/presentation/shillaker

This paper is included in the Proceedings of the
2020 USENIX Annual Technical Conference.

July 15–17, 2020

978-1-939133-14-4

Open access to the Proceedings of the
2020 USENIX Annual Technical Conference
is sponsored by USENIX.

FAASM: Lightweight Isolation for Efficient Stateful Serverless Computing

Simon Shillaker
Imperial College London

Peter Pietzuch
Imperial College London

Abstract

Serverless computing is an excellent fit for big data processing because it can scale quickly and cheaply to thousands of parallel functions. Existing serverless platforms isolate functions in ephemeral, stateless containers, preventing them from directly sharing memory. This forces users to duplicate and serialise data repeatedly, adding unnecessary performance and resource costs. We believe that a new lightweight isolation approach is needed, which supports sharing memory directly between functions and reduces resource overheads.

We introduce *Faaslets*, a new isolation abstraction for high-performance serverless computing. Faaslets isolate the memory of executed functions using *software-fault isolation* (SFI), as provided by *WebAssembly*, while allowing memory regions to be shared between functions in the same address space. Faaslets can thus avoid expensive data movement when functions are co-located on the same machine. Our runtime for Faaslets, FAASM, isolates other resources, e.g. CPU and network, using standard Linux *cgroups*, and provides a low-level POSIX host interface for networking, file system access and dynamic loading. To reduce initialisation times, FAASM restores Faaslets from already-initialised snapshots. We compare FAASM to a standard container-based platform and show that, when training a machine learning model, it achieves a 2× speed-up with 10× less memory; for serving machine learning inference, FAASM doubles the throughput and reduces tail latency by 90%.

1 Introduction

Serverless computing is becoming a popular way to deploy data-intensive applications. A function-as-a-service (FaaS) model decomposes computation into many functions, which can effectively exploit the massive parallelism of clouds. Prior work has shown how serverless can support map/reduce-style jobs [42, 69], machine learning training [17, 18] and inference [40], and linear algebra computation [73, 88]. As a result, an increasing number of applications, implemented in diverse programming languages, are being migrated to serverless platforms.

Existing platforms such as Google Cloud Functions [32], IBM Cloud Functions [39], Azure Functions [50] and AWS Lambda [5] isolate functions in ephemeral, stateless *containers*. The use of containers as an isolation mechanisms introduces two challenges for data-intensive applications, *data access overheads* and the *container resource footprint*.

Data access overheads are caused by the stateless nature of the container-based approach, which forces state to be maintained externally, e.g. in object stores such as Amazon S3 [6], or passed between function invocations. Both options incur costs due to duplicating data in each function, repeated serialisation, and regular network transfers. This results in current applications adopting an inefficient "data-shipping architecture", i.e. moving data to the computation and not vice versa—such architectures have been abandoned by the data management community many decades ago [36]. These overheads are compounded as the number of functions increases, reducing the benefit of unlimited parallelism, which is what makes serverless computing attractive in the first place.

The container resource footprint is particularly relevant because of the high-volume and short-lived nature of serverless workloads. Despite containers having a smaller memory and CPU overhead than other mechanisms such as virtual machines (VMs), there remains an impedance mismatch between the execution of individual short-running functions and the process-based isolation of containers. Containers have start-up latencies in the hundreds of milliseconds to several seconds, leading to the *cold-start* problem in today's serverless platforms [36, 83]. The large memory footprint of containers limits scalability—while technically capped at the process limit of a machine, the maximum number of containers is usually limited by the amount of available memory, with only a few thousand containers supported on a machine with 16 GB of RAM [51].

Current data-intensive serverless applications have addressed these problems individually, but never solved both—instead, either exacerbating the container resource overhead or breaking the serverless model. Some systems avoid data movement costs by maintaining state in long-lived VMs or ser-

vices, such as ExCamera [30], Shredder [92] and Cirrus [18], thus introducing non-serverless components. To address the performance overhead of containers, systems typically increase the level of trust in users' code and weaken isolation guarantees. PyWren [42] reuses containers to execute multiple functions; Crucial [12] shares a single instance of the Java virtual machine (JVM) between functions; SAND [1] executes multiple functions in long-lived containers, which also run an additional message-passing service; and Cloudburst [75] takes a similar approach, introducing a local key-value-store cache. Provisioning containers to execute multiple functions and extra services amplifies resource overheads, and breaks the fine-grained elastic scaling inherent to serverless. While several of these systems reduce data access overheads with local storage, none provide *shared memory* between functions, thus still requiring duplication of data in separate process memories.

Other systems reduce the container resource footprint by moving away from containers and VMs. Terrarium [28] and Cloudflare Workers [22] employ software-based isolation using WebAssembly and V8 Isolates, respectively; Krustlet [54] replicates containers using WebAssembly for memory safety; and SEUSS [16] demonstrates serverless unikernels. While these approaches have a reduced resource footprint, they do not address data access overheads, and the use of software-based isolation alone does not isolate resources.

We make the observation that serverless computing can better support data-intensive applications with a new isolation abstraction that (i) provides strong memory and resource isolation between functions, yet (ii) supports efficient state sharing. Data should be *co-located* with functions and accessed directly, minimising data-shipping. Furthermore, this new isolation abstraction must (iii) allow scaling state across multiple hosts; (iv) have a low memory footprint, permitting many instances on one machine; (v) exhibit fast instantiation times; and (vi) support multiple programming languages to facilitate the porting of existing applications.

In this paper, we describe **Faaslets**, a new *lightweight isolation abstraction* for data-intensive serverless computing. Faaslets support stateful functions with efficient shared memory access, and are executed by our **FAASM** distributed serverless runtime. Faaslets have the following properties, summarising our contributions:

(1) Faaslets achieve lightweight isolation. Faaslets rely on *software fault isolation* (SFI) [82], which restricts functions to accesses of their own memory. A function associated with a Faaslet, together with its library and language runtime dependencies, is compiled to WebAssembly [35]. The FAASM runtime then executes multiple Faaslets, each with a dedicated thread, within a single address space. For resource isolation, the CPU cycles of each thread are constrained using Linux *cgroups* [79] and network access is limited using *network namespaces* [79] and *traffic shaping*. Many Faaslets can be executed efficiently and safely on a single machine.

(2) Faaslets support efficient local/global state access. Since Faaslets share the same address space, they can access shared memory regions with local state efficiently. This allows the co-location of data and functions and avoids serialisation overheads. Faaslets use a two-tier state architecture, a *local* tier provides in-memory sharing, and a *global* tier supports distributed access to state across hosts. The FAASM runtime provides a state management API to Faaslets that gives fine-grained control over state in both tiers. Faaslets also support stateful applications with different consistency requirements between the two tiers.

(3) Faaslets have fast initialisation times. To reduce cold-start time when a Faaslet executes for the first time, it is launched from a suspended state. The FAASM runtime pre-initialises a Faaslet ahead-of-time and snapshots its memory to obtain a *Proto-Faaslet*, which can be restored in hundreds of microseconds. Proto-Faaslets are used to create fresh Faaslet instances quickly, e.g. avoiding the time to initialise a language runtime. While existing work on snapshots for serverless takes a single-machine approach [1, 16, 25, 61], Proto-Faaslets support cross-host restores and are OS-independent.

(4) Faaslets support a flexible host interface. Faaslets interact with the host environment through a set of POSIX-like calls for networking, file I/O, global state access and library loading/linking. This allows them to support dynamic language runtimes and facilitates the porting of existing applications, such as CPython by changing fewer than 10 lines of code. The host interface provides just enough virtualisation to ensure isolation while adding a negligible overhead.

The FAASM runtime[1] uses the LLVM compiler toolchain to translate applications to WebAssembly and supports functions written in a range of programming languages, including C/C++, Python, Typescript and Javascript. It integrates with existing serverless platforms, and we describe the use with *Knative* [33], a state-of-the-art platform based on Kubernetes.

To evaluate FAASM's performance, we consider a number of workloads and compare to a container-based serverless deployment. When training a machine learning model with SGD [68], we show that FAASM achieves a 60% improvement in run time, a 70% reduction in network transfers, and a 90% reduction in memory usage; for machine learning inference using TensorFlow Lite [78] and MobileNet [37], FAASM achieves over a 200% increase in maximum throughput, and a 90% reduction in tail latency. We also show that FAASM executes a distributed linear algebra job for matrix multiplication using Python/Numpy with negligible performance overhead and a 13% reduction in network transfers.

2 Isolation vs. Sharing in Serverless

Sharing memory is fundamentally at odds with the goal of isolation, hence providing shared access to in-memory state

[1] FAASM is open-source and available at github.com/lsds/Faasm

		Containers	VMs	Unikernel	SFI	Faaslet
Func.	Memory safety	✓	✓	✓	✓	✓
	Resource isolation	✓	✓	✓	✗	✓
	Efficient state sharing	✗	✗	✗	✗	✓
	Shared filesystem	✓	✗	✗	✓	✓
Non-func.	Initialisation time	100 ms	100 ms	10 ms	10 µs	1 ms
	Memory footprint	MBs	MBs	KBs	Bytes	KBs
	Multi-language	✓	✓	✓	✗	✓

Table 1: Isolation approaches for serverless (Initialisation times include ahead-of-time snapshot restore where applicable [16,25,61].)

in a multi-tenant serverless environment is a challenge.

Tab. 1 contrasts *containers* and *VMs* with other potential serverless isolation options, namely *unikernels* [16] in which minimal VM images are used to pack tasks densely on a hypervisor and *software-fault isolation* (SFI) [82], providing lightweight memory safety through static analysis, instrumentation and runtime traps. The table lists whether each fulfils three key functional requirements: memory safety, resource isolation and sharing of in-memory state. A fourth requirement is the ability to share a filesystem between functions, which is important for legacy code and to reduce duplication with shared files.

The table also compares these options on a set of non-functional requirements: low initialisation time for fast elasticity; small memory footprint for scalability and efficiency, and the support for a range of programming languages.

Containers offer an acceptable balance of features if one sacrifices efficient state sharing—as such they are used by many serverless platforms [32, 39, 50]. Amazon uses Firecracker [4], a "micro VM" based on KVM with similar properties to containers, e.g. initialisation times in the hundreds of milliseconds and memory overheads of megabytes.

Containers and VMs compare poorly to unikernels and SFI on initialisation times and memory footprint because of their level of virtualisation. They both provide complete virtualised POSIX environments, and VMs also virtualise hardware. Unikernels minimise their levels of virtualisation, while SFI provides none. Many unikernel implementations, however, lack the maturity required for production serverless platforms, e.g. missing the required tooling and a way for non-expert users to deploy custom images. SFI alone cannot provide resource isolation, as it purely focuses on memory safety. It also does not define a way to perform isolated interactions with the underlying host. Crucially, as with containers and VMs, neither unikernels nor SFI can share state efficiently, with no way to express shared memory regions between compartments.

2.1 Improving on Containers

Serverless functions in containers typically share state via external storage and duplicate data across function instances. Data access and serialisation introduces network and compute overheads; duplication bloats the memory footprint of containers, already of the order of megabytes [51]. Containers contribute hundreds of milliseconds up to seconds in cold-start latencies [83], incurred on initial requests and when scaling. Existing work has tried to mitigate these drawbacks by recycling containers between functions, introducing static VMs, reducing storage latency, and optimising initialisation.

Recycling containers avoids initialisation overheads and allows data caching but sacrifices isolation and multi-tenancy. PyWren [42] and its descendants, Numpywren [73], IBMPywren [69], and Locus [66] use recycled containers, with long-lived AWS Lambda functions that dynamically load and execute Python functions. Crucial [12] takes a similar approach, running multiple functions in the same JVM. SAND [1] and Cloudburst [75] provide only process isolation between functions of the same application and place them in shared long-running containers, with at least one additional background storage process. Using containers for multiple functions and supplementary long-running services requires over-provisioned memory to ensure capacity both for concurrent executions and for peak usage. This is at odds with the idea of fine-grained scaling in serverless.

Adding static VMs to handle external storage improves performance but breaks the serverless paradigm. Cirrus [18] uses large VM instances to run a custom storage back-end; Shredder [92] uses a single long-running VM for both storage and function execution; ExCamera [30] uses long-running VMs to coordinate a pool of functions. Either the user or provider must scale these VMs to match the elasticity and parallelism of functions, which adds complexity and cost.

Reducing the latency of auto-scaled storage can improve performance within the serverless paradigm. Pocket [43] provides ephemeral serverless storage; other cloud providers offer managed external state, such as AWS Step Functions [3], Azure Durable Functions [53], and IBM Composer [8]. Such approaches, however, do not address the data-shipping problem and its associated network and memory overheads.

Container initialisation times have been reduced to mitigate the cold-start problem, which can contribute several seconds of latency with standard containers [36, 72, 83]. SOCK [61] improves the container boot process to achieve cold starts in the low hundreds of milliseconds; Catalyzer [25] and SEUSS [16] demonstrate snapshot and restore in VMs and unikernels to achieve millisecond serverless cold starts. Although such reductions are promising, the resource overhead and restrictions on sharing memory in the underlying mechanisms still remain.

2.2 Potential of Software-based Isolation

Software-based isolation offers memory safety with initialisation times and memory overheads up to two orders of magnitude lower than containers and VMs. For this reason, it is an attractive starting point for serverless isolation. However, software-based isolation alone does not support resource isolation, or efficient in-memory state sharing.

It has been used in several existing edge and serverless computing systems, but none address these shortcomings.

Fastly's Terrarium [28] and Cloudflare Workers [22] provide memory safety with WebAssembly [35] and V8 Isolates [34], respectively, but neither isolates CPU or network use, and both rely on data shipping for state access; Shredder [92] also uses V8 Isolates to run code on a storage server, but does not address resource isolation, and relies on co-locating state and functions on a single host. This makes it ill-suited to the level of scale required in serverless platforms; Boucher et al. [14] show microsecond initialisation times for Rust microservices, but do not address isolation or state sharing; Krustlet [54] is a recent prototype using WebAssembly to replace Docker in Kubernetes, which could be integrated with Knative [33]. It focuses, however, on replicating container-based isolation, and so fails to meet our requirement for in-memory sharing.

Our final non-functional requirement is for multi-language support, which is not met by language-specific approaches to software-based isolation [11, 27]. Portable Native Client [23] provides multi-language software-based isolation by targeting a portable intermediate representation, LLVM IR, and hence meets this requirement. Portable Native Client has now been deprecated, with WebAssembly as its successor [35].

WebAssembly offers strong memory safety guarantees by constraining memory access to a single linear byte array, referenced with offsets from zero. This enables efficient bounds checking at both compile- and runtime, with runtime checks backed by traps. These traps (and others for referencing invalid functions) are implemented as part of WebAssembly runtimes [87]. The security guarantees of WebAssembly are well established in existing literature, which covers formal verification [84], taint tracking [31], and dynamic analysis [45]. WebAssembly offers mature support for languages with an LLVM front-end such as C, C++, C#, Go and Rust [49], while toolchains exist for Typescript [10] and Swift [77]. Java bytecode can also be converted [7], and further language support is possible by compiling language runtimes to WebAssembly, e.g. Python, JavaScript and Ruby. Although WebAssembly is restricted to a 32-bit address space, 64-bit support is in development.

The WebAssembly specification does not yet include mechanisms for sharing memory, therefore it alone cannot meet our requirements. There is a proposal to add a form of synchronised shared memory to WebAssembly [85], but it is not well suited to sharing serverless state dynamically due to the required compile-time knowledge of all shared regions. It also lacks an associated programming model and provides only local memory synchronisation.

The properties of software-based isolation highlight a compelling alternative to containers, VMs and unikernels, but none of these approaches meet all of our requirements. We therefore propose a new isolation approach to enable efficient serverless computing for big data.

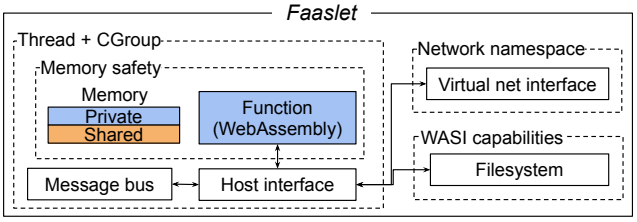


Figure 1: Faaslet abstraction with isolation

3 Faaslets

We propose *Faaslets*, a new isolation mechanism that satisfies all the requirements for efficient data-intensive serverless computing. Tab. 1 highlights Faaslets' strong memory and resource isolation guarantees, and efficient *shared in-memory state*. Faaslets provide a minimal level of lightweight virtualisation through their *host interface*, which supports serverless-specific tasks, memory management, a limited filesystem and network access.

In terms of non-functional requirements, Faaslets improve on containers and VMs by having a memory footprint below 200 KB and cold-start initialisation times of less than 10 ms. Faaslets execute functions compiled to secure IR, allowing them to support multiple programming languages.

While Faaslets cannot initialise as quickly as pure SFI, they mitigate the cold-start problem through ahead-of-time initialisation from snapshots called *Proto-Faaslets*. Proto-Faaslets reduce initialisation times to hundreds of microseconds, and a single snapshot can be restored across hosts, quickly scaling horizontally on clusters.

3.1 Overview

Fig. 1 shows a function isolated inside a Faaslet. The function itself is compiled to WebAssembly [35], guaranteeing memory safety and control flow integrity. By default, a function is placed in its own *private* contiguous memory region, but Faaslets also support *shared regions* of memory (§3.3). This allows a Faaslet to access shared in-memory state within the constraints of WebAssembly's memory safety guarantees.

Faaslets also ensure fair resource access. For CPU isolation, they use the CPU subset of Linux *cgroups* [79]. Each function is executed by a dedicated *thread* of a shared runtime process. This thread is assigned to a cgroup with a share of CPU equal to that of all Faaslets. The Linux CFS [79] ensures that these threads are scheduled with equal CPU time.

Faaslets achieve secure and fair network access using *network namespaces*, *virtual network interfaces* and *traffic shaping* [79]. Each Faaslet has its own network interface in a separate namespace, configured using *iptables* rules. To ensure fairness between co-located tenants, each Faaslet applies traffic shaping on its virtual network interface using `tc`, thus enforcing ingress and egress traffic rate limits.

As functions in a Faaslet must be permitted to invoke standard system calls to perform memory management and I/O operations, Faaslets offer an interface through which to in-

Class	Function	Action	Standard
Calls	`byte* read_call_input()`	Read input data to function as byte array	*none*
	`void write_call_output(out_data)`	Write output data for function	
	`int chain_call(name, args)`	Call function and return the `call_id`	
	`int await_call(call_id)`	Await the completion of `call_id`	
	`byte* get_call_output(call_id)`	Load the output data of `call_id`	
State	`byte* get_state(key, flags)`	Get pointer to state value for `key`	
	`byte* get_state_offset(key, off, flags)`	Get pointer to state value for `key` at offset	
	`void set_state(key, val)`	Set state value for `key`	
	`void set_state_offset(key, val, len, off)`	Set `len` bytes of state value at offset for `key`	
	`void push/pull_state(key)`	Push/pull global state value for `key`	
	`void push/pull_state_offset(key, off)`	Push/pull global state value for `key` at offset	
	`void append_state(key, val)`	Append data to state value for `key`	
	`void lock_state_read/write(key)`	Lock local copy of state value for `key`	
	`void lock_state_global_read/write(key)`	Lock state value for `key` globally	
Dynlink	`void* dlopen/dlsym(...)`	Dynamic linking of libraries	POSIX
	`int dlclose(...)`	*As above*	
Memory	`void* mmap(...)`, `int munmap(...)`	Memory grow/shrink only	
	`int brk(...)`, `void* sbrk(...)`	Memory grow/shrink	
Network	`int socket/connect/bind(...)`	Client-side networking only	WASI
	`size_t send/recv(...)`	Send/recv via virtual interface	
File I/O	`int open/close/dup/stat(...)`	Per-user virtual filesystem access	
	`size_t read/write(...)`	*As above*	
Misc	`int gettime(...)`	Per-user monotonic clock only	
	`size_t getrandom(...)`	Uses underlying host `/dev/urandom`	

Table 2: Faaslet host interface (The final column indicates whether functions are defined as part of POSIX or WASI [57].)

teract with the underlying host. Unlike containers or VMs, Faaslets do not provide a fully-virtualised POSIX environment but instead support a minimal serverless-specific host interface (see Fig. 1). Faaslets virtualise system calls that interact with the underlying host and expose a range of functionality, as described below.

The host interface integrates with the serverless runtime through a *message bus* (see Fig. 1). The message bus is used by Faaslets to communicate with their parent process and each other, receive function calls, share work, invoke and await other functions, and to be told by their parent process when to spawn and terminate.

Faaslets support a *read-global write-local* filesystem, which lets functions read files from a global object store (§5), and write to locally cached versions of the files. This is primarily used to support legacy applications, notably language runtimes such as CPython [67], which need a filesystem for loading library code and storing intermediate bytecode. The filesystem is accessible through a set of POSIX-like API functions that implement the WASI capability-based security model, which provides efficient isolation through unforgeable file handles [56]. This removes the need for more resource-intensive filesystem isolation such as a layered filesystem or `chroot`, which otherwise add to cold start latencies [61].

3.2 Host Interface

The Faaslet host interface must provide a virtualisation layer capable of executing a range of serverless big data applications, as well as legacy POSIX applications. This interface necessarily operates outside the bounds of memory safety, and hence is trusted to preserve isolation when interacting with the host.

In existing serverless platforms based on containers and VMs, this virtualisation layer is a standard POSIX environment, with serverless-specific tasks executed through language- and provider-specific APIs over HTTP [5, 32, 39]. Instantiating a full POSIX environment with the associated isolation mechanisms leads to high initialisation times [61], and heavy use of HTTP APIs contributes further latency and network overheads.

In contrast, the Faaslet host interface targets minimal virtualisation, hence reducing the overheads required to provide isolation. The host interface is a low-level API built exclusively to support a range of high-performance serverless applications. The host interface is dynamically linked with function code at runtime (§3.4), making calls to the interface more efficient than performing the same tasks through an external API.

Tab. 2 lists the Faaslet host interface API, which supports: (i) chained serverless function invocation; (ii) interacting with shared state (§4); (iii) a subset of POSIX-like calls for memory management, timing, random numbers, file/network I/O and dynamic linking. A subset of these POSIX-like calls are implemented according to WASI, an emerging standard for a server-side WebAssembly interface [57]. Some key details of the API are as follows:

Function invocation. Functions retrieve their input data serialised as byte arrays using the `read_call_input` function, and similarly write their output data as byte arrays using `write_call_output`. Byte arrays constitute a generic, language-agnostic interface.

Non-trivial serverless applications invoke multiple func-

tions that work together as part of chained calls, made with the `chain_call` function. Users' functions have unique names, which are passed to `chain_call`, along with a byte array containing the input data for that call.

A call to `chain_call` returns the call ID of the invoked function. The call ID can then be passed to `await_call` to perform a blocking wait for another call to finish or fail, yielding its return code. The Faaslet blocks until the function has completed, and passes the same call ID to `get_call_output` to retrieve the chained call's output data.

Calls to `chain_call` and `await_call` can be used in loops to spawn and await calls in a similar manner to standard multi-threaded code: one loop invokes `chain_call` and records the call IDs; a second loop calls `await_call` on each ID in turn. We show this pattern in Python in Listing 1.

Dynamic linking. Some legacy applications and libraries require support for dynamic linking, e.g. CPython dynamically links Python extensions. All dynamically loaded code must first be compiled to WebAssembly and undergo the same validation process as other user-defined code (§3.4). Such modules are loaded via the standard Faaslet filesystem abstraction and covered by the same safety guarantees as its parent function. Faaslets support this through a standard POSIX dynamic linking API, which is implemented according to WebAssembly dynamic linking conventions [86].

Memory. Functions allocate memory dynamically through calls to `mmap()` and `brk()`, either directly or through `dlmalloc` [44]. The Faaslet allocates memory in its private memory region, and uses `mmap` on the underlying host to extend the region if necessary. Each function has its own predefined memory limit, and these calls fail if growth of the private region would exceeded this limit.

Networking. The supported subset of networking calls allows simple client-side send/receive operations and is sufficient for common use cases, such as connecting to an external data store or a remote HTTP endpoint. The functions `socket`, `connect` and `bind` allow setting up the socket while `read` and `write` allow the sending and receiving of data. Calls fail if they pass flags that are not related to simple send/receive operations over IPv4/IPv6, e.g. the `AF_UNIX` flag.

The host interface translates these calls to equivalent socket operations on the host. All calls interact exclusively with the Faaslet's virtual network interface, thus are constrained to a private network interface and cannot exceed rate limits due to the traffic shaping rules.

Byte arrays. Function inputs, results and state are represented as simple byte arrays, as is all function memory. This avoids the need to serialise and copy data as it passes through the API, and makes it trivial to share arbitrarily complex in-memory data structures.

3.3 Shared Memory Regions

As discussed in §2, sharing in-memory state while otherwise maintaining isolation is an important requirement for efficient serverless big data applications. Faaslets do this by adding the new concept of *shared regions* to the existing WebAssembly memory model [35]. Shared regions give functions concurrent access to disjoint segments of shared process memory, allowing them direct, low-latency access to shared data structures. Shared regions are backed by standard OS virtual memory, so there is no extra serialisation or overhead, hence Faaslets achieve efficient concurrent access on a par with native multi-threaded applications. In §4.2, we describe how Faaslets use this mechanism to provide shared in-memory access to global state.

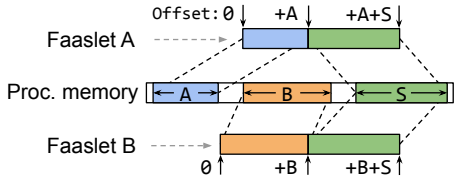


Figure 2: Faaslet shared memory region mapping

Shared regions maintain the memory safety guarantees of the existing WebAssembly memory model, and use standard OS virtual memory mechanisms. WebAssembly restricts each function's memory to a contiguous linear byte array, which is allocated by the Faaslet at runtime from a disjoint section of the process memory. To create a new shared region, the Faaslet extends the function's linear byte array, and remaps the new pages onto a designated region of common process memory. The function accesses the new region of linear memory as normal, hence maintaining memory safety, but the underlying memory accesses are mapped onto the shared region.

Fig. 2 shows Faaslets A and B accessing a shared region (labelled S), allocated from a disjoint region of the common process memory (represented by the central region). Each Faaslet has its own region of private memory (labelled A and B), also allocated from the process memory. Functions inside each Faaslet access all memory as offsets from zero, forming a single linear address space. Faaslets map these offsets onto either a private region (in this case the lower offsets), or a shared region (in this case the higher offsets).

Multiple shared regions are permitted, and functions can also extend their private memory through calls to the memory management functions in the host interface such as `brk` (§3.2). Extension of private memory and creation of new shared regions is handled by extending a byte array, which represents the function's memory, and then remapping the underlying pages to regions of shared process memory. This means the function continues to see a single densely-packed linear address space, which may be backed by several virtual memory mappings. Faaslets allocate shared process memory through calls to `mmap` on the underlying host, passing `MAP_SHARED` and `MAP_ANONYMOUS` flags to create shared and private regions, respectively, and remap these regions with `mremap`.

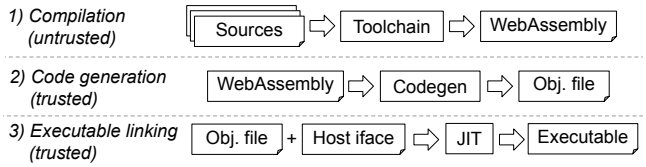


Figure 3: Creation of a Faaslet executable

3.4 Building Functions for Faaslets

Fig. 3 shows the three phases to convert source code of a function into a Faaslet executable: (1) the user invokes the Faaslet toolchain to compile the function into a WebAssembly binary, linking against a language-specific declaration of the Faaslet host interface; (2) code generation creates an object file with machine code from WebAssembly; and (3) the host interface definition is linked with the machine code to produce the Faaslet executable.

When Faaslets are deployed, the compilation phase to generate the WebAssembly binary takes place on a user's machine. Since that is untrusted, the code generation phase begins by validating the WebAssembly binary, as defined in the WebAssembly specification [35]. This ensures that the binary conforms to the specification. Code generation then takes place in a trusted environment, after the user has uploaded their function.

In the linking phase, the Faaslet uses LLVM JIT libraries [49] to link the object file and the definition of the host interface implementation. The host interface functions are defined as *thunks*, which allows injecting the trusted host interface implementation into the function binary.

Faaslets use WAVM [70] to perform the validation, code generation and linking. WAVM is an open-source WebAssembly VM, which passes the WebAssembly conformance tests [84] and thus guarantees that the resulting executable enforces memory safety and control flow integrity [35].

4 Local and Global State

Stateful serverless applications can be created with Faaslets using *distributed data objects (DDO)*, which are language-specific classes that expose a convenient high-level state interface. DDOs are implemented using the key/value state API from Tab. 2.

The state associated with Faaslets is managed using a *two-tier* approach that combines local sharing with global distribution of state: a *local tier* provides shared in-memory access to state on the same host; and a *global tier* allows Faaslets to synchronise state across hosts.

DDOs hide the two-tier state architecture, providing transparent access to distributed data. Functions, however, can still access the state API directly, either to exercise more fine-grained control over consistency and synchronisation, or to implement custom data structures.

Listing 1: Distributed SGD application with Faaslets

```
1  t_a = SparseMatrixReadOnly("training_a")
2  t_b = MatrixReadOnly("training_b")
3  weights = VectorAsync("weights")
4
5  @faasm_func
6  def weight_update(idx_a, idx_b):
7    for col_idx, col_a in t_a.columns[idx_a:idx_b]:
8      col_b = t_b.columns[col_idx]
9      adj = calc_adjustment(col_a, col_b)
10     for val_idx, val in col_a.non_nulls():
11       weights[val_idx] += val * adj
12       if iter_count % threshold == 0:
13         weights.push()
14
15 @faasm_func
16 def sgd_main(n_workers, n_epochs):
17   for e in n_epochs:
18     args = divide_problem(n_workers)
19     c = chain(update, n_workers, args)
20     await_all(c)
21     ...
```

4.1 State Programming Model

Each DDO represents a single state value, referenced throughout the system using a string holding its respective state key.

Faaslets write changes from the local to the global tier by performing a *push*, and read from the global to the local tier by performing a *pull*. DDOs may employ push and pull operations to produce variable consistency, such as delaying updates in an eventually-consistent list or set, and may lazily pull values only when they are accessed, such as in a distributed dictionary. Certain DDOs are immutable, and hence avoid repeated synchronisation.

Listing 1 shows both implicit and explicit use of two-tier state through DDOs to implement stochastic gradient descent (SGD) in Python. The `weight_update` function accesses two large input matrices through the `SparseMatrixReadOnly` and `MatrixReadOnly` DDOs (lines 1 and 2), and a single shared weights vector using `VectorAsync` (line 3). `VectorAsync` exposes a `push()` function which is used to periodically push updates from the local tier to the global tier (line 13). The calls to `weight_update` are chained in a loop in `sgd_main` (line 19).

Function `weight_update` accesses a randomly assigned subset of columns from the training matrices using the `columns` property (lines 7 and 8). The DDO implicitly performs a pull operation to ensure that data is present, and only replicates the necessary subsets of the state values in the local tier—the entire matrix is not transferred unnecessarily.

Updates to the shared weights vector in the local tier are made in a loop in the `weight_update` function (line 11). It invokes the `push` method on this vector (line 13) sporadically to update the global tier. This improves performance and reduces network overhead, but introduces inconsistency between the tiers. SGD tolerates such inconsistencies and it does not affect the overall result.

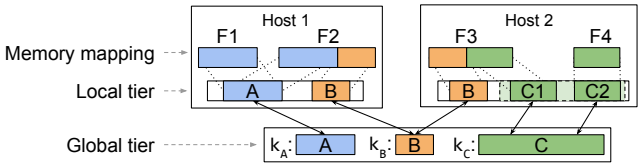


Figure 4: Faaslet two-tier state architecture

4.2 Two-Tier State Architecture

Faaslets represent state with a key/value abstraction, using unique *state keys* to reference *state values*. The authoritative state value for each key is held in the global tier, which is backed by a distributed key-value store (KVS) and accessible to all Faaslets in the cluster. Faaslets on a given host share a local tier, containing replicas of each state value currently mapped to Faaslets on that host. The local tier is held exclusively in Faaslet shared memory regions, and Faaslets do not have a separate local storage service, as in SAND [1] or Cloudburst [75].

Fig. 4 shows the two-tier state architecture across two hosts. Faaslets on host 1 share state value A; Faaslets on both hosts share state value B. Accordingly, there is a replica of state value A in the local tier of host 1, and replicas of state value B in the local tier of both hosts.

The `columns` method of the `SparseMatrixReadOnly` and `MatrixReadOnly` DDOs in Listing 1 uses *state chunks* to access a subset of a larger state value. As shown in Fig. 4, state value C has state chunks, which are treated as smaller independent state values. Faaslets create replicas of only the required chunks in their local tier.

Ensuring local consistency. State value replicas in the local tier are created using Faaslet shared memory (§3.3). To ensure consistency between Faaslets accessing a replica, Faaslets acquire a *local read lock* when reading, and a *local write lock* when writing. This locking happens implicitly as part of all state API functions, but not when functions write directly to the local replica via a pointer. The state API exposes the `lock_state_read` and `lock_state_write` functions that can be used to acquire local locks explicitly, e.g. to implement a list that performs multiple writes to its state value when atomically adding an element. A Faaslet creates a new local replica after a call to `pull_state` or `get_state` if it does not already exist, and ensures consistency through a write lock.

Ensuring global consistency. DDOs can produce varying levels of consistency between the tiers as shown by `VectorAsync` in Listing 1. To enforce strong consistency, DDOs must use *global read/write locks*, which can be acquired and released for each state key using `lock_state_global_read` and `lock_state_global_write`, respectively. To perform a consistent write to the global tier, an object acquires a global write lock, calls `pull_state` to update the local tier, applies its write to the local tier, calls `push_state` to update the global tier, and releases the lock.

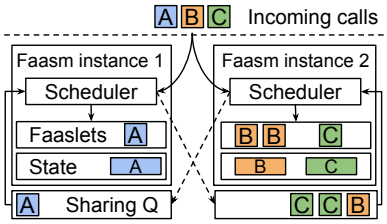


Figure 5: FAASM system architecture

5 FAASM Runtime

FAASM is the serverless runtime that uses Faaslets to execute distributed stateful serverless applications across a cluster. FAASM is designed to integrate with existing serverless platforms, which provide the underlying infrastructure, auto-scaling functionality and user-facing frontends. FAASM handles the scheduling, execution and state management of Faaslets. The design of FAASM follows a distributed architecture: multiple FAASM runtime instances execute on a set of servers, and each instance manages a pool of Faaslets.

5.1 Distributed Scheduling

A local scheduler in the FAASM runtime is responsible for the scheduling of Faaslets. Its scheduling strategy is key to minimising data-shipping (see §2) by ensuring that executed functions are co-located with required in-memory state. One or more Faaslets managed by a runtime instance may be *warm*, i.e. they already have their code and state loaded. The scheduling goal is to ensure that as many function calls as possible are executed by warm Faaslets.

To achieve this without modifications to the underlying platform's scheduler, FAASM uses a *distributed shared state* scheduler similar to Omega [71]. Function calls are sent round-robin to local schedulers, which execute the function locally if they are warm and have capacity, or share it with another warm host if one exists. The set of warm hosts for each function is held in the FAASM state global tier, and each scheduler may query and atomically update this set during the scheduling decision.

Fig. 5 shows two FAASM runtime instances, each with its own local scheduler, a pool of Faaslets, a collection of state stored in memory, and a sharing queue. Calls for functions A–C are received by the local schedulers, which execute them locally if they have warm Faaslets, and share them with the other host if not. Instance 1 has a warm Faaslet for function A and accepts calls to this function, while sharing calls to functions B and C with Instance 2, which has corresponding warm Faaslets. If a function call is received and there are no instances with warm Faaslets, the instance that received the call creates a new Faaslet, incurring a "cold start".

5.2 Reducing Cold Start Latency

While Faaslets typically initialise in under 10 ms, FAASM reduces this further using *Proto-Faaslets*, which are Faaslets that contain snapshots of arbitrary execution state that can be restored on any host in the cluster. From this snapshot,

FAASM spawns a new Faaslet instance, typically reducing initialisation to hundreds of microseconds (§6.5).

Different Proto-Faaslets are generated for a function by specifying user-defined *initialisation code*, which is executed before snapshotting. If a function executes the same code on each invocation, that code can become initialisation code and be removed from the function itself. For Faaslets with dynamic language runtimes, the runtime initialisation can be done as part of the initialisation code.

A Proto-Faaslet snapshot includes a function's stack, heap, function table, stack pointer and data, as defined in the WebAssembly specification [35]. Since WebAssembly memory is represented by a contiguous byte array, containing the stack,heap and data, FAASM restores a snapshot into a new Faaslet using a copy-on-write memory mapping. All other data is held in standard C++ objects. Since the snapshot is independent of the underlying OS thread or process, FAASM can serialise Proto-Faaslets and instantiate them across hosts.

FAASM provides an *upload* service that exposes an HTTP endpoint. Users upload WebAssembly binaries to this endpoint, which then performs code generation (§3.4) and writes the resulting object files to a shared *object store*. The implementation of this store is specific to the underlying serverless platform but can be a cloud provider's own solution such as AWS S3 [6]. Proto-Faaslets are generated and stored in the FAASM global state tier as part of this process. When a Faaslet undergoes a cold start, it loads the object file and Proto-Faaslet, and restores it.

In addition, FAASM uses Proto-Faaslets to reset Faaslets after each function call. Since the Proto-Faaslet captures a function's initialised execution state, restoring it guarantees that no information from the previous call is disclosed. This can be used for functions that are *multi-tenant*, e.g. in a serverless web application. FAASM guarantees that private data held in memory is cleared away after each function execution, thereby allowing Faaslets to handle subsequent calls across tenants. In a container-based platform, this is typically not safe, as the platform cannot ensure that the container memory has been cleaned entirely between calls.

6 Evaluation

Our experimental evaluation targets the following questions: (i) how does FAASM state management improve efficiency and performance on parallel machine learning training? (§6.2) (ii) how do Proto-Faaslets and low initialisation times impact performance and throughput in inference serving? (§6.3) (iii) how does Faaslet isolation affect performance in a linear algebra benchmark using a dynamic language runtime? (§6.4) and (iv) how do the overheads of Faaslets compare to Docker containers? (§6.5)

6.1 Experimental Set-up

Serverless baseline. To benchmark FAASM against a state-of-the-art serverless platform, we use Knative [33], a container-based system built on Kubernetes [80]. All experiments are implemented using the same code for both FAASM and Knative, with a Knative-specific implementation of the Faaslet host interface for container-based code. This interface uses the same undelrying state management code as FAASM, but cannot share the local tier between co-located functions. Knative function chaining is performed through the standard Knative API. Redis is used for the distributed KVS and deployed to the same cluster.

FAASM integration. We integrate FAASM with Knative by running FAASM runtime instances as Knative functions that are replicated using the default autoscaler. The system is otherwise unmodified, using the default endpoints and scheduler.

Testbed. Both FAASM and Knative applications are executed on the same Kubernetes cluster, running on 20 hosts, all Intel Xeon E3-1220 3.1 GHz machines with 16 GB of RAM, connected with a 1 Gbps connection. Experiments in §6.5 were run on a single Intel Xeon E5-2660 2.6 GHz machine with 32 GB of RAM.

Metrics. In addition to the usual evaluation metrics, such as execution time, throughput and latency, we also consider *billable memory*, which quantifies memory consumption over time. It is the product of the peak function memory multiplied by the number and runtime of functions, in units of GB-seconds. It is used to attribute memory usage in many serverless platforms [5, 32, 39]. Note that all memory measurements include the containers/Faaslets and their state.

6.2 Machine Learning Training

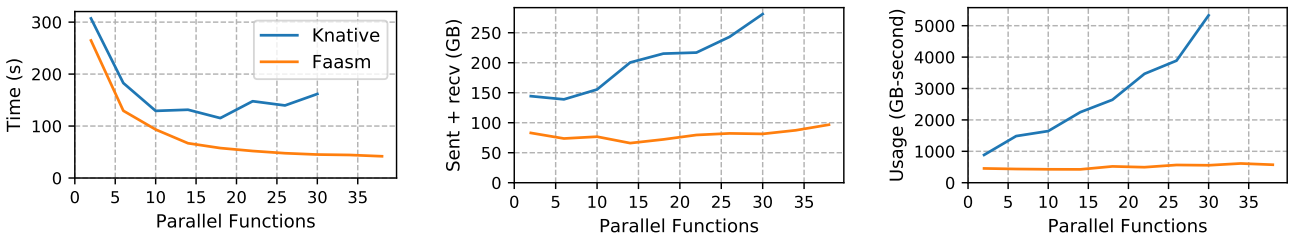
This experiment focuses on the impact of FAASM's state management on runtime, network overheads and memory usage.

We use distributed *stochastic gradient descent* (SGD) using the HOGWILD! algorithm [68] to run text classification on the Reuters RCV1 dataset [46]. This updates a central weights vector in parallel with batches of functions across multiple epochs. We run both Knative and FAASM with increasing numbers of parallel functions.

Fig. 6a shows the training time. FAASM exhibits a small improvement in runtime of 10% compared to Knative at low parallelism and a 60% improvement with 15 parallel functions. With more than 20 parallel Knative functions, the underlying hosts experience increased memory pressure and they exhaust memory with over 30 functions. Training time continues to improve for FAASM up to 38 parallel functions, at which point there is a more than an 80% improvement over 2 functions.

Fig. 6b shows that, with increasing parallelism, the volume of network transfers increases in both FAASM and Knative. Knative transfers more data to start with and the volume increase more rapidly, with 145 GB transferred with 2 parallel functions and 280 GB transferred with 30 functions. FAASM transfers 75 GB with 2 parallel functions and 100 GB with 38 parallel functions.

Fig. 6c shows that billable memory in Knative increases with more parallelism: from 1,000 GB-seconds for 2 func-



(a) Training time (b) Network transfers (c) Memory usage

Figure 6: Machine learning training with SGD with Faaslets (FAASM) and containers (Knative)

tions to over 5,000 GB-second for 30 functions. The billable memory for FAASM increases slowly from 350 GB-second for 2 functions to 500 GB-second with 38 functions.

The increased network transfer, memory usage and duration in Knative is caused primarily by data shipping, e.g. loading data into containers. FAASM benefits from sharing data through its local tier, hence amortises overheads and reduces latency. Further improvements in duration and network overhead come from differences in the updates to the shared weights vector: in FAASM, the updates from multiple functions are batched per host; whereas in Knative, each function must write directly to external storage. Billable memory in Knative and FAASM increases with more parallelism, however, the increased memory footprint and duration in Knative make this increase more pronounced.

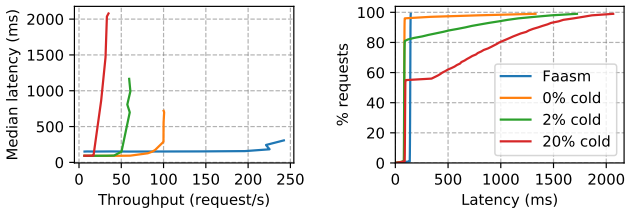
To isolate the underlying performance and resource overheads of FAASM and Knative, we run the same experiment with the number of training examples reduced from 800K to 128. Across 32 parallel functions, we observe for FAASM and Knative: training times of 460 ms and 630 ms; network transfers of 19 MB and 48 MB; billable memory usage of 0.01 GB-second and 0.04 GB-second, respectively.

In this case, increased duration in Knative is caused by the latency and volume of inter-function communication through the Knative HTTP API versus direct inter-Faaslet communication. FAASM incurs reduced network transfers versus Knative as in the first experiment, but the overhead of these transfers in both systems are negligible as they are small and amortized across all functions. Billable memory is increased in Knative due to the memory overhead of each function container being 8 MB (versus 270 kB for each Faaslet). These improvements are negligible when compared with those derived from reduced data shipping and duplication of the full dataset.

6.3 Machine Learning Inference

This experiment explores the impact of the Faaslet initialisation times on cold-starts and function call throughput.

We consider a machine learning inference application because they are typically user-facing, thus latency-sensitive, and must serve high volumes of requests. We perform inference serving with TensorFlow Lite [78], with images loaded from a file server and classified using a pre-trained MobileNet [37] model. In our implementation, requests from



(a) Throughput vs. latency (b) Latency CDF

Figure 7: Machine learning inference with TensorFlow Lite

each user are sent to different instances of the underlying serverless function. Therefore, each user sees a cold-start on their first request. We measure the latency distribution and change in median latency when increasing throughput and varying the ratio of cold-starts.

Figs. 7a and 7b show a single line for FAASM that covers all cold-start ratios. Cold-starts only introduce a negligible latency penalty of less than 1 ms and do not add significant resource contention, hence all ratios behave the same. Optimal latency in FAASM is higher than that in Knative, as the inference calculation takes longer due to the performance overhead from compiling TensorFlow Lite to WebAssembly.

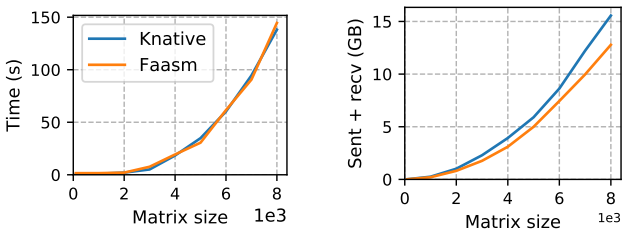
Fig. 7a shows that the median latency in Knative increases sharply from a certain throughput threshold depending on the cold-start ratio. This is caused by cold starts resulting in queuing and resource contention, with the median latency for the 20% cold-start workload increasing from 90 ms to over 2 s at around 20 req/s. FAASM maintains a median latency of 120 ms at a throughput of over 200 req/s.

Fig. 7b shows the latency distribution for a single function that handles successive calls with different cold-start ratios. Knative has a tail latency of over 2 s and more than 35% of calls have latencies of over 500 ms with 20% cold-starts. FAASM achieves a tail latency of under 150 ms for all ratios.

6.4 Language Runtime Performance with Python

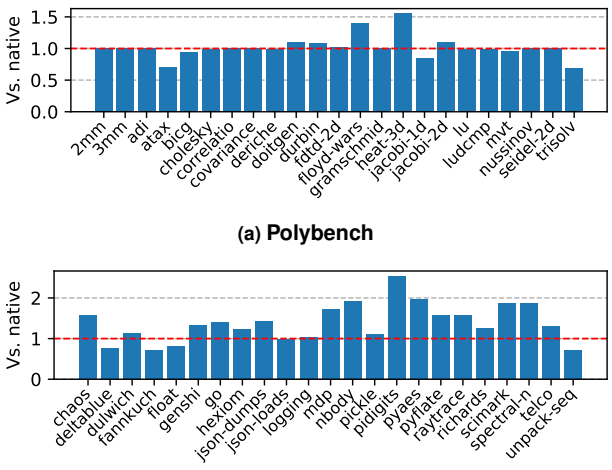
The next two experiments (i) measure the performance impact of Faaslet isolation on a distributed benchmark using an existing dynamic language runtime, the CPython interpreter; and (ii) investigate the impact on a single Faaslet running compute microbenchmarks and a suite of Python microbenchmarks.

We consider a distributed divide-and-conquer matrix multiplication implemented with Python and Numpy. In the FAASM implementation, these functions are executed using



(a) Duration (b) Network transfer

Figure 8: Comparison of matrix multiplication with Numpy



(a) Polybench

(b) Python Performance Benchmark

Figure 9: Performance of Faaslets with Python

CPython inside a Faaslet; in Knative, we use standard Python. As there is no WebAssembly support for BLAS and LAPACK, we do not use them in either implementation.

While this experiment is computationally intensive, it also makes use of the filesystem, dynamic linking, function chaining and state, thus exercising all of the Faaslet host interface. Each matrix multiplication is subdivided into multiplications of smaller submatrices and merged. This is implemented by recursively chaining serverless functions, with each multiplication using 64 multiplication functions and 9 merging functions. We compare the execution time and network traffic when running multiplications of increasingly large matrices.

Fig. 8a shows that the duration of matrix multiplications on FAASM and Knative are almost identical with increasing matrix sizes. Both take around 500 ms with 100×100 matrices, and almost 150 secs with 8000×8000 matrices. Fig. 8b shows that FAASM results in 13% less network traffic across all matrix sizes, and hence gains a small benefit from storing intermediate results more efficiently.

In the next experiment, we use Polybench/C [64] to measure the Faaslet performance overheads on simple compute functions, and the Python Performance Benchmarks [76] for overheads on more complex applications. Polybench/C is compiled directly to WebAssembly and executed in Faaslets; the Python code executes with CPython running in a Faaslet.

	Docker	Faaslets	Proto-Faaslets	vs. Docker
Initialisation	2.8 s	5.2 ms	0.5 ms	**5.6K**×
CPU cycles	251M	1.4K	650	**385K**×
PSS memory	1.3 MB	200 KB	90 KB	**15**×
RSS memory	5.0 MB	200 KB	90 KB	**57**×
Capacity	~8 K	~70 K	>100 K	**12**×

Table 3: Comparison of Faaslets vs. container cold starts (no-op function)

Fig. 9 shows the performance overhead when running both sets of benchmarks compared to native execution. All but two of the Polybench benchmarks are comparable to native with some showing performance gains. Two experience a 40%–55% overhead, both of which benefit from loop optimisations that are lost through compilation to WebAssembly. Although many of the Python benchmarks are within a 25% overhead or better, some see a 50%–60% overhead, with `pidigits` showing a 240% overhead. `pidigits` stresses big integer arithmetic, which incurs significant overhead in 32-bit WebAssembly.

Jangda et al. [41] report that code compiled to WebAssembly has more instructions, branches and cache misses, and that these overheads are compounded on larger applications. Serverless functions, however, typically are not complex applications and operate in a distributed setting in which distribution overheads dominate. As shown in Fig. 8a, FAASM can achieve competitive performance with native execution, even for functions interpreted by a dynamic language runtime.

6.5 Efficiency of Faaslets vs. Containers

Finally we focus on the difference in footprint and cold-start initialisation latency between Faaslets and containers.

To measure memory usage, we deploy increasing numbers of parallel functions on a host and measure the change in footprint with each extra function. Containers are built from the same minimal image (`alpine:3.10.1`) so can access the same local copies of shared libraries. To highlight the impact of this sharing, we include the proportional set size (PSS) and resident set size (RSS) memory consumption. Initialisation times and CPU cycles are measured across repeated executions of a no-op function. We observe the capacity as the maximum number of concurrent running containers or Faaslets that a host can sustain before running out of memory.

Tab. 3 shows several orders of magnitude improvement in CPU cycles and time elapsed when isolating a no-op with Faaslets, and a further order of magnitude using Proto-Faaslets. With an optimistic PSS memory measurement for containers, memory footprints are almost seven times lower using Faaslets, and 15× lower using Proto-Faaslets. A single host can support up to 10× more Faaslets than containers, growing to twelve times more using Proto-Faaslets.

To assess the impact of restoring a non-trivial Proto-Faaslet snapshot, we run the same initialisation time measurement for a Python no-op function. The Proto-Faaslet snapshot is a pre-initialised CPython interpreter, and the container uses a minimal `python:3.7-alpine` image. The container

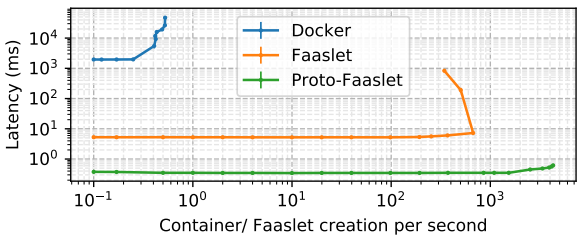


Figure 10: Function churn for Faaslets vs. containers

initialises in 3.2 s and the Proto-Faaslet restores in 0.9 ms, demonstrating a similar improvement of several orders of magnitude.

To further investigate cold-start initialisation times, we measure the time to create a new container/Faaslet at increasingly higher rates of cold-starts per second. We also measure this time when restoring the Faaslet from a Proto-Faaslet. The experiment executes on a single host, with the containers using the same minimal image.

Fig. 10 shows that both Faaslets and containers maintain a steady initialisation latency at throughputs below 3 execution/s, with Docker containers initialising in ~2 s and Faaslets in ~5 ms (or ~0.5 ms when restored from a Proto-Faaslet). As we increase the churn in Docker past 3 execution/s, initialisation times begin to increase with no gain in throughput. A similar limit for Faaslets is reached at around 600 execution/s, which grows to around 4000 execution/s with Proto-Faaslets.

We conclude that Faaslets offer a more efficient and performant form of serverless isolation than Docker containers, which is further improved with Proto-Faaslets. The lower resource footprint and initialisation times of Faaslets are important in a serverless context. Lower resource footprints reduce costs for the cloud provider and allow a higher packing density of parallel functions on a given host. Low initialisation times reduce cost and latency for the user, through their mitigation of the cold-start problem.

7 Related Work

Isolation mechanisms. Shreds [20] and Wedge [13] introduce new OS-level primitives for memory isolation, but focus on intra-process isolation rather than a complete executable as Faaslets do. Light-weight Contexts [48] and Picoprocesses [38] offer lightweight sandboxing of complete POSIX applications, but do not offer efficient shared state.

Common runtimes. Truffle [90] and GraalVM [26] are runtimes for language-independent bytecode; the JVM also executes multiple languages compiled to Java bytecode [21]. Despite compelling multi-language support, none offer multi-tenancy or resource isolation. GraalVM has recently added support for WebAssembly and could be adapted for Faaslets.

Autoscaling storage. FAASM's global state tier is currently implemented with a distributed Redis instance scaled by Kubernetes horizontal pod autoscaler [81]. Although this has not been a bottleneck, better alternatives exist: Anna [89] is a distributed KVS that achieves lower latency and more granular autoscaling than Redis; Tuba [9] provides an autoscaling KVS that operates within application-defined constraints; and Pocket [43] is a granular autoscaled storage system built specifically for a serverless environments. Crucial [12] uses Infinispan [52] to build its distributed object storage, which could also be used to implement FAASM's global state tier.

Distributed shared memory (DSM). FaRM [24] and RAMCloud [63] demonstrate that fast networks can overcome the historically poor performance of DSM systems [19], while DAL [60] demonstrates the benefits of introducing locality awareness to DSM. FAASM's global tier could be replaced with DSM to form a distributed object store, which would require a suitable consensus protocol, such as Raft [62], and a communication layer, such as Apache Arrow [65].

State in distributed dataflows. Spark [91] and Hadoop [74] support stateful distributed computation. Although focuses on fixed-size clusters and not fine-grained elastic scaling or multi-tenancy, distributed dataflow systems such as Naiad [58], SDGs [29] and CIEL [59] provide high-level interfaces for distributed state, with similar aims to those of distributed data objects—they could be implemented in or ported to FAASM. Bloom [2] provides a high-level distributed programming language, focused particularly on flexible consistency and replication, ideas also relevant to FAASM.

Actor frameworks. Actor-based systems such as Orleans [15], Akka [47] and Ray [55] support distributed stateful tasks, freeing users from scheduling and state management, much like FAASM. However, they enforce a strict asynchronous programming model and are tied to a specific languages or language runtimes, without multi-tenancy.

8 Conclusions

To meet the increasing demand for serverless big data, we presented FAASM, a runtime that delivers high-performance efficient state without compromising isolation. FAASM executes functions inside Faaslets, which provide memory safety and resource fairness, yet can share in-memory state. Faaslets are initialised quickly thanks to Proto-Faaslet snapshots. Users build stateful serverless applications with distributed data objects on top of the Faaslet state API. FAASM's two-tier state architecture co-locates functions with required state, providing parallel in-memory processing yet scaling across hosts. The Faaslet host interface also supports dynamic language runtimes and traditional POSIX applications.

Acknowledgements. We thank the anonymous reviewers and our shepherd, Don Porter, for their valuable feedback. This work was partially supported by the European Union's Horizon 2020 Framework Programme under grant agreement 825184 (CloudButton), the UK Engineering and Physical Sciences Research Council (EPSRC) award 1973141, and a gift from Intel Corporation under the TFaaS project.

References

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

[2] Peter Alvaro, Neil Conway, Joseph M. Hellerstein, and William R. Marczak. Consistency Analysis in Bloom: A CALM and Collected Approach. In *Conference on Innovative Data Systems Research (CIDR)*, 2011.

[3] Amazon. AWS Step Functions. https://aws.amazon.com/step-functions/, 2020.

[4] Amazon. Firecracker Micro VM. https://aws.amazon.com/blogs/aws/firecracker-lightweight-virtualization-for-serverless-computing/, 2020.

[5] Amazon Web Services. AWS Lambda. https://aws.amazon.com/lambda/, 2020.

[6] Amazon Web Services. AWS S3. https://aws.amazon.com/s3/, 2020.

[7] Alexey Andreev. TeaVM. http://www.teavm.org/, 2020.

[8] Apache Project. Openwhisk Composer. https://github.com/ibm-functions/composer, 2020.

[9] Masoud Saeida Ardekani and Douglas B Terry. A Self-Configurable Geo-Replicated Cloud Storage System. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[10] Assemblyscript. AssemblyScript. https://github.com/AssemblyScript/assemblyscript, 2020.

[11] Abhiram Balasubramanian, Marek S Baranowski, Anton Burtsev, Aurojit Panda, Zvonimir Rakamari, and Leonid Ryzhyk. System Programming in Rust: Beyond Safety. *ACM SIGOPS Operating Systems Review*, 2017.

[12] Daniel Barcelona-Pons, Marc Sánchez-Artigas, Gerard París, Pierre Sutra, and Pedro García-López. On the FaaS Track: Building Stateful Distributed Applications with Serverless Architectures. In *ACM/IFIP Middleware Conference*, 2019.

[13] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. Wedge: Splitting Applications into Reduced-Privilege Compartments. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2008.

[14] Sol Boucher, Anuj Kalia, David G Andersen, and Michael Kaminsky. Putting the "Micro" Back in Microservice. *USENIX Annual Technical Conference (USENIX ATC)*, 2018.

[15] Sergey Bykov, Alan Geller, Gabriel Kliot, James R Larus, Ravi Pandya, and Jorgen Thelin. Orleans: Cloud Computing for Everyone. In *ACM Symposium on Cloud Computing (SOCC)*, 2011.

[16] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. SEUSS: Skip Redundant Paths to Make Serverless Fast. In *ACM European Conference on Computer Systems (EuroSys)*, 2020.

[17] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy Katz. A Case for Serverless Machine Learning. *Systems for ML*, 2018.

[18] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy Katz. Cirrus. In *ACM Symposium on Cloud Computing (SOCC)*, 2019.

[19] John B Carter, John K Bennett, and Willy Zwaenepoel. Implementation and Performance of Munin. *ACM SIGOPS Operating Systems Review*, 1991.

[20] Y Chen, S Reymondjohnson, Z Sun, and L Lu. Shreds: Fine-Grained Execution Units with Private Memory. In *IEEE Symposium on Security and Privacy (SP)*, 2016.

[21] Shigeru Chiba and Muga Nishizawa. An Easy-to-use Toolkit for Efficient Java Bytecode Translators. In *International Conference on Generative Programming and Component Engineering*, 2003.

[22] Cloudflare. Cloudflare Workers. https://workers.cloudflare.com/, 2020.

[23] Alan Donovan, Robert Muth, Brad Chen, and David Sehr. Pnacl: Portable Native Client Executables. *Google White Paper*, 2010.

[24] Aleksandar Dragojevic, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast Remote Memory. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2014.

[25] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. Catalyzer: Sub-millisecond Startup for Serverless Computing with Initialization-less Booting. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[26] Gilles Duboscq, Lukas Stadler, Thomas Würthinger, Doug Simon, Christian Wimmer, and Hanspeter Mössenböck. Graal IR: An Extensible Declarative Intermediate Representation. In *Proceedings of the Asia-Pacific Programming Languages and Compilers Workshop*, 2013.

[27] Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen Hunt, James R Larus, and Steven Levi. Language Support for Fast and Reliable Message-based Communication in Singularity OS. In *ACM SIGOPS Operating Systems Review*, 2006.

[28] Fastly. Terrarium. https://wasm.fastlylabs.com/, 2020.

[29] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. Making State Explicit For Imperative Big Data Processing. In *USENIX Annual Technical Conference (USENIX ATC)*, 2014.

[30] Sadjad Fouladi, Riad S Wahby, Brennan Shacklett, Karthikeyan Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads. *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.

[31] William Fu, Raymond Lin, and Daniel Inge. TaintAssembly: Taint-Based Information Flow Control Tracking for WebAssembly. *arXiv preprint arXiv:1802.01050*, 2018.

[32] Google. Google Cloud Functions. https://cloud.google.com/functions/, 2020.

[33] Google. KNative Github. https://github.com/knative, 2020.

[34] Google. V8 Engine. https://github.com/v8/v8, 2020.

[35] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the Web up to Speed with WebAssembly. *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2017.

[36] Joseph M. Hellerstein, Jose Faleiro, Joseph E. Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. Serverless Computing: One Step Forward, Two Steps Back. *Conference on Innovative Data Systems Research (CIDR)*, 2019.

[37] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *arXiv preprint arXiv:1704.04861*, 2017.

[38] Jon Howell, Bryan Parno, and John R Douceur. How to Run POSIX Apps in a Minimal Picoprocess. In *USENIX Annual Technical Conference (USENIX ATC)*, 2013.

[39] IBM. IBM Cloud Functions. https://www.ibm.com/cloud/functions, 2020.

[40] Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. Serving Deep Learning Models in a Serverless Platform. In *IEEE International Conference on Cloud Engineering, (IC2E)*, 2018.

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

[42] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. Occupy the Cloud: Distributed Computing for the 99%. In *ACM Symposium on Cloud Computing (SOCC)*, 2017.

[43] Ana Klimovic, Yawen Wang, Stanford University, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. Pocket: Elastic Ephemeral Storage for Serverless Analytics. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[44] Doug Lea. dlmalloc. http://gee.cs.oswego.edu/dl/html/malloc.html, 2020.

[45] Daniel Lehmann and Michael Pradel. Wasabi: A Framework for Dynamically Analyzing WebAssembly. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[46] David D. Lewis, Yiming Yang, Tony G. Rose, and Fan Li. RCV1: A New Benchmark Collection for Text Categorization Research. *Journal of Machine Learning Research*, 2004.

[47] Lightbend. Akka Framework. https://akka.io/, 2020.

[48] James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, Bobby Bhattacharjee, and Peter Druschel. Light-Weight Contexts: An OS Abstraction for Safety and Performance. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[49] LLVM Project. LLVM 9 Release Notes. https://releases.llvm.org/9.0.0/docs/ReleaseNotes.html, 2020.

[50] Sahil Malik. Azure Functions. https://azure.microsoft.com/en-us/services/functions/, 2020.

[51] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. My VM is Lighter (and Safer) than your Container. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

[52] Francesco Marchioni and Manik Surtani. *Infinispan Data Grid Platform*. Packt Publishing Ltd, 2012.

[53] Microsoft. Azure Durable Functions. https://docs.microsoft.com/en-us/azure/azure-functions/durable-functions-overview, 2020.

[54] Microsoft Research. Krustlet. https://deislabs.io/posts/introducing-krustlet/.

[55] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A Distributed Framework for Emerging AI Applications. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2017.

[56] Mozilla. WASI Design Principles. https://github.com/WebAssembly/WASI/blob/master/docs/DesignPrinciples.md, 2020.

[57] Mozilla. WASI: WebAssembly System Interface. https://wasi.dev/, 2020.

[58] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: a Timely Dataflow System. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[59] Derek G Murray, Malte Schwarzkopf, Christopher Smowton, Steven Smith, Anil Madhavapeddy, and Steven Hand. CIEL: a Universal Execution Engine for Distributed Dataflow Computing. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.

[60] Gábor Németh, Dániel Géhberger, and Péter Mátray. DAL: A Locality-Optimizing Distributed Shared Memory System. In *USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2017.

[61] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. SOCK: Rapid Task Provisioning with Serverless-Optimized Containers. In *USENIX Annual Technical Conference (USENIX ATC)*, 2018.

[62] Diego Ongaro and John Ousterhout. In Search of an Understandable Consensus Algorithm. In *USENIX Annual Technical Conference (USENIX ATC)*, 2014.

[63] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, Stephen Rumble, Ryan Stutsman, and Stephen Yang. The RAMCloud Storage System. *ACM Transactions on Computer Systems*, 2015.

[64] Louis-Noel Pouchet. Polybench/C. http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/, 2020.

[65] The Apache Project. Apache arrow. https://arrow.apache.org/.

[66] Qifan Pu, U C Berkeley, Shivaram Venkataraman, Ion Stoica, U C Berkeley, and Implementation Nsdi. Shuffling, fast and slow : Scalable analytics on serverless infrastructure. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2019.

[67] Python Software Foundation. CPython. https://github.com/python/cpython, 2020.

[68] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild: A Lock-free Approach to Parallelizing Stochastic Gradient Descent. In *Advances in Neural Information Processing Systems*, 2011.

[69] Josep Sampé, Gil Vernik, Marc Sánchez-Artigas, and Pedro García-López. Serverless Data Analytics in the IBM Cloud. In *ACM/IFIP Middleware Conference*, 2018.

[70] Andrew Scheidecker. WAVM. https://github.com/WAVM/WAVM, 2020.

[71] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. Omega: Flexible, Scalable Schedulers for Large Compute Clusters. In *ACM European Conference on Computer Systems (EuroSys)*, 2013.

[72] Mohammad Shahrad, Jonathan Balkind, and David Wentzlaff. Architectural Implications of Function-as-a-service Computing. In *Annual International Symposium on Microarchitecture (MICRO)*, 2019.

[73] Vaishaal Shankar, Karl Krauth, Qifan Pu, Eric Jonas, Shivaram Venkataraman, Ion Stoica, Benjamin Recht, and Jonathan Ragan-Kelley. Numpywren: Serverless Linear Algebra. *arXivpreprint arXiv:1810.09679*, 2018.

[74] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, Robert Chansler, and Others. The Hadoop Distributed File System. In *Conference on Massive Storage Systems and Technology (MSST)*, 2010.

[75] Vikram Sreekanti, Chenggang Wu Xiayue Charles Lin, Jose M Faleiro, Joseph E Gonzalez, Joseph M Hellerstein, and Alexey Tumanov. Cloudburst: Stateful Functions-as-a-Service. *arXiv preprint arXiv:2001.04592*, 2020.

[76] Victor Stinner. The Python Performance Benchmark Suite. https://pyperformance.readthedocs.io/, 2020.

[77] SwiftWasm. SwiftWasm. https://swiftwasm.org/.

[78] Tensorflow. TensorFlow Lite. https://www.tensorflow.org/lite, 2020.

[79] The Kernel Development Community. The Linux Kernel documentation. https://www.kernel.org/doc/html/v4.10/driver-api/80211/mac80211.html, 2020.

[80] The Linux Foundation. Kubernetes. https://kubernetes.io/, 2020.

[81] The Linux Foundation. Kubernetes Horizontal Pod Autoscaler. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/, 2020.

[82] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient Software-based Fault Isolation. In *ACM Symposium on Operating Systems Principles (SOSP)*, 1993.

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

[84] Conrad Watt. Mechanising and Verifying the WebAssembly Specification. In *ACM SIGPLAN International Conference on Certified Programs and Proofs*, 2018.

[85] Conrad Watt, Andreas Rossberg, and Jean Pichon-Pharabod. Weakening WebAssembly. *Proceedings of the ACM on Programming Languages (PACMPL)*, 2019.

[86] WebAssembly. WebAssembly Dynamic Linking. https://webassembly.org/docs/dynamic-linking/, 2020.

[87] WebAssembly. WebAssembly Specification. https://github.com/WebAssembly/spec/, 2020.

[88] S. Werner, J. Kuhlenkamp, M. Klems, J. Müller, and S. Tai. Serverless Big Data Processing Using Matrix Multiplication. In *IEEE Conference on Big Data (Big Data)*, 2018.

[89] Chenggang Wu, Jose Faleiro, Yihan Lin, and Joseph Hellerstein. Anna: a KVS for any Scale. *IEEE International Conference on Data Engineering, (ICDE)*, 2018.

[90] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. One VM to Rule Them All. In *ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, 2013.

[91] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2012.

[92] Tian Zhang, Dong Xie, Feifei Li, and Ryan Stutsman. Narrowing the Gap Between Serverless and its State with Storage Functions. In *ACM Symposium on Cloud Computing (SOCC)*, 2019.