Function as a Function

Tom Kuchler
ETH Zurich
Zurich, Switzerland
kuchlert@ethz.ch

Michael Giardino[*]
ETH Zurich
Zurich, Switzerland
mgiardino@ethz.ch

Timothy Roscoe
ETH Zurich
Zurich, Switzerland
troscoe@ethz.ch

Ana Klimovic
ETH Zurich
Zurich, Switzerland
aklimovic@ethz.ch

ABSTRACT

Function as a Service (FaaS) and the associated serverless computing paradigm alleviates users from resource management and allows cloud platforms to optimize system infrastructure under the hood. Despite significant advances, FaaS infrastructure still leaves much room to improve performance and resource efficiency. We argue that both higher performance and resource efficiency are possible — while maintaining secure isolation — if we are willing to revisit the FaaS programming model and system software design. We propose *Dandelion*, a clean-slate FaaS system that rethinks the programming model by treating serverless functions as pure functions, thereby explicitly separating computation and I/O. This new programming model enables a lightweight yet secure function execution system. It also makes functions more amenable to hardware acceleration and enables dataflow-aware function orchestration. Our initial prototype of Dandelion achieves 45× lower tail latency for cold starts compared to Firecracker. For 95% hot function invocations, Dandelion achieves 5× higher peak throughput.

CCS CONCEPTS

• **Computer systems organization** → **Cloud computing**; • **Software and its engineering** → **Cloud computing**.

KEYWORDS

serverless, cloud computing, function as a service

ACM Reference Format:
Tom Kuchler, Michael Giardino, Timothy Roscoe, and Ana Klimovic. 2023. Function as a Function. In *ACM Symposium on Cloud Computing (SoCC '23), October 30–November 1, 2023, Santa Cruz, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3620678.3624648

[*]Currently with Computing Systems Lab, Huawei Technologies, Zurich.

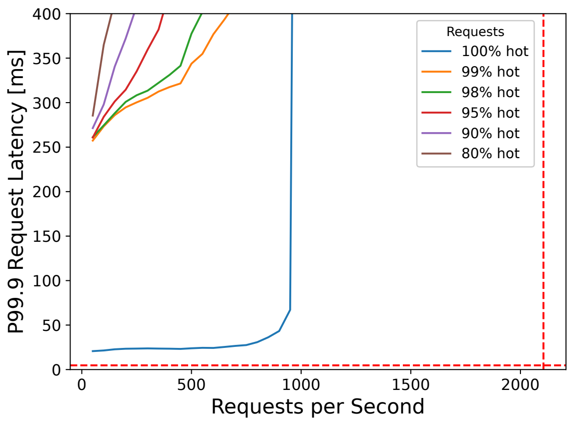

Figure 1: Round-trip tail latency for remote function execution with Firecracker, varying % hot requests. Red dotted lines show local bare-metal function execution latency (horizontal) and peak throughput (vertical).

1 INTRODUCTION

Serverless computing has the potential to become the dominant paradigm of cloud computing [58, 15], making cloud facilities easier to use and enabling cloud providers to more transparently optimize performance and energy efficiency of their infrastructure. With serverless, users develop applications as compositions of fine-grained functions, which execute independently while having access to shared remote storage. Users invoke functions on-demand and the cloud platform dynamically allocates the necessary hardware resources to execute them with an appealing pay-for-what-you-use cost model.

While this model holds promise, the system software infrastructure it uses is still rooted in the very different, more traditional execution model of long-running processes or virtual machines. Cloud providers typically provide function isolation by running them inside separate 'lightweight' VMs, which still incur significant startup times [62], context switch overheads [66], and memory duplication [56]. This practice of bundling each function with its own OS leads to a very general API, and the need to support this makes it hard for cloud providers to efficiently use their resources to run functions with low latency.

To quantify the performance and energy efficiency left on the table by current FaaS system software, we run an experiment using Firecracker [2] as an example of a state-of-the-art FaaS hypervisor. AWS Lambda uses Firecracker to run functions inside MicroVMs, which have significantly lower startup time than traditional VMs. In Figure 1, we measure round-trip response time for a simple matrix multiplication function invoked over HTTP and executed on a Firecracker server running on a 10-core Intel Xeon E5-2640v4. We use a simple HTTP frontend to relay function invocation requests to Firecracker, which executes functions in MicroVMs. Although simplified, our setup captures the essential functionality of FaaS worker nodes. To estimate the cost of executing this function on the remote Firecracker server compared to locally, the horizontal dashed red line in Figure 1 shows the function's local bare-metal execution time (4.75 ms) and the vertical dashed line shows the corresponding peak throughput. Remotely executing the function significantly reduces peak throughput (and hence energy efficiency), even when all invocations go to hot (already booted) Firecracker MicroVMs, as seen in the 100% hot curve. As we increase the percentage of cold starts (i.e., requests for which a new MicroVM must be booted), tail latency increases significantly and saturates at far lower throughput. Although there are unavoidable networking overheads compared to local bare-metal function execution, this experiment demonstrates that state-of-the-art FaaS system software still has significant overhead, especially when booting a function sandbox is on the critical path. While prior work has optimized the unloaded startup latency of function sandboxes (e.g., by restoring state from snapshots [19, 62, 9]), function execution still suffers at high request loads and with high churn of function sandboxes. This current approach of retrofitting existing system software, which is still rooted in the model of executing long-running virtual machines, misses the opportunity to harness the true potential of the serverless computing paradigm.

In this paper, we argue for a different execution environment better suited to serverless, which permits much more efficient resource usage and reduced startup latency, while still supporting the use-cases that make serverless computing so attractive. Our model is to *treat serverless functions as true functions*: bodies of code which take a declared list of input parameters (e.g. data sets on stable storage) and output another, declared list of output data sets which can be fed to other functions. During execution, a function performs no I/O - indeed, it barely needs to invoke any services from system software.

This clear separation of computation (the function itself) and I/O (now handled completely outside the function, logically before or after its execution), immediately leads to many opportunities to improve *both* the performance of functions and the efficiency with which they can be supported by the cloud platform. Moreover, the delegation of I/O to cloud-implemented functions can provide better isolation, obviating the need to ensure safety of the broad range of user-issued system calls. More concretely, moving all I/O and other system software dependencies out of the function allows using simplified lightweight thread-like sandboxes which can leverage a range of different hardware protection mechanisms for secure isolation, such as CHERI memory capabilities [68], MMUs, etc.

Scheduling is also greatly simplified, since functions do not block, and start running when their input is guaranteed to be available. Dependencies between functions (and their data) are more explicit, enabling the platform to further optimize placement and scheduling. Functions can now also be replaced with alternative implementations that generate the same output for a given input, making it possible to transparently use hardware accelerators when available.

We realize this model in Dandelion, a platform for serverless function execution. We prototype Dandelion's function execution system. In benchmarks, our prototype running on Arm's Morello hardware platform [8] using CHERI memory capabilities [68] for memory isolation, achieves 45× lower tail latency for cold starts and reaches a 5× higher peak throughput for 95% hot request load compared to Firecracker. These gains come from replacing the current bloated worker node software stack with a lightweight system that leverages modern hardware to improve performance and energy efficiency, without sacrificing security.

2 FAAS PROPERTIES AND REQUIREMENTS

The serverless computing execution model — in which the cloud platform automatically manages and scales resources to execute user code based on request load — is appealing for many applications, from event-driven web services to data analytics [21, 42, 16].

Serverless functions have unique characteristics. They have *short execution time*, often less than a second [59, 63] with some platforms [16] seeing median execution time of only 60 ms. Functions generally have *small resource footprints*, with a median memory allocation of only 170 MB per function [59]. FaaS applications tend to have more *bursty load* than traditional cloud workloads [28, 20]. For example, the peak-to-trough ratio of function invocation can be as high as 500× [63]. Finally, *invocations are sporadic*, with fewer than 20% invoked more than once per minute [59].

The characteristics of FaaS workloads lead to the following requirements for FaaS platforms:

Low end-to-end function execution latency: A function should complete with minimal overhead compared to its execution on a dedicated, bare-metal server.

High throughput per CAPEX: To maximize throughput per capital expenditure, FaaS system software should serve a high rate of function execution requests per server to maximize utilization.

Energy efficiency: To minimize operational expenses — particularly energy consumption — the FaaS system should minimize CPU cycles for scheduling and executing functions.

Secure isolation: FaaS system software must prevent untrusted user function code from tampering with the infrastructure or accessing the data or code of other functions. Our threat model assumes that *users trust the cloud provider*, as is customary today [4].

3 THE CURRENT STATE OF SERVERLESS

We discuss state-of-the-art system infrastructure for FaaS function execution and scheduling, highlighting why current solutions do not fully satisfy the requirements in §2. Prior work has mainly focused on minimizing unloaded function latency, but optimizing throughput and energy efficiency without sacrificing security is particularly challenging.

3.1 Secure Isolation of Functions

Today's FaaS platforms isolate functions by implementing sandboxes with one of three key techniques: virtualization, containerization, or language runtime isolation.

Virtualization: Most commercial FaaS platforms rely on virtualization to execute and isolate functions [40]. Specialized MicroVMs [2] greatly reduce startup times compared to general-purpose VMs, but still have significant overheads, particularly with high sandbox churn. MicroVMs also increase a function's memory footprint (e.g., 3× memory overhead for a function with a 1 MB working set [57]), which limits the degree of function co-location and thus the throughput per worker node.

Snapshot restoration [61, 62, 45, 6, 64, 19, 18] and Unikernel-based VMs [35, 14, 30, 37, 39] reduce startup delays down to millisecond-range. However, snapshots introduce security issues with random number generator state and address space layout randomization [13] and neither of these approaches solves FaaS performance issues at high request load and sandbox churn.

Containerization: Some FaaS platforms execute functions in containers [24, 46, 32]. Containers rely on OS primitives for resource isolation. Since the OS interface is a known source of security vulnerabilities [23], most major cloud providers do not consider traditional containers sufficiently secure isolation for FaaS. gVisor [24] improves container security by adding a software interposition layer, however its performance is similar to MicroVMs [7].

Runtime isolation: To further reduce sandbox initialization and memory overheads, researchers have proposed non-virtualized sandboxes, such as processes [10, 60, 11] and isolation via language runtimes [12, 61, 65]. For instance, WebAssembly runs user code in a sandboxed environment, with the compiler or interpreter inserting runtime checks, restricting the code to its own memory region [27]. Language runtime isolation approaches trust the runtime to correctly set up and tear down sandboxes and implement the system interface that programs use to read files, send network requests, and access other OS resources [26]. This can be problematic as it involves error-prone low-level memory management, and bugs in the runtime can break isolation guarantees. Formally verifying language runtime and system interfaces is an active area of research [26], but end-to-end system verification is challenging.

3.2 Function Scheduling and Data Passing

Traditional FaaS platforms are oblivious to the communication patterns and data dependencies between functions [25]. This simplifies scheduling as the platform treats each function as a black box, but forces functions to interact with remote storage to exchange data, which adds latency and cost [31, 48, 23, 33].

Cloud providers have started offering services, such as AWS Step Functions [3] and Azure Durable Functions [41], which allow users to chain functions and express their dependencies. However, specifying a dependency in these systems is mainly a hint to spin up function sandboxes, rather than a way to optimize data transfer between them. Intermediate data that cannot be attached to an invocation request still must be transferred via storage.

Pheromone [72] proposes a data-centric approach to function orchestration with a new API of data trigger primitives. It currently relies on memory sharing for efficient implementation, which can be problematic in a public cloud environment where securely isolating untrusted user code is a strict requirement. Faa$t [53] and Palette [1] apply caching to minimize interaction with remote storage, but they rely on repeated requests from the same user or concurrently running sandboxes. Boxer [69] enables functions to establish direct TCP connections with other functions, avoiding round-trips to storage. However, its programming model does not reveal function dependencies upfront to the platform, limiting opportunities for data dependency-aware scheduling. Deng *et al.* [17] propose separating compute and I/O to enable reproducible serverless computations with correctness guarantees. In §4, we will discuss how separating compute and I/O enables a step change in FaaS platform efficiency.

3.3 Heterogeneous Hardware Support

FaaS system software has not kept up with the tremendous advances in cloud hardware over the past decade, such as energy-efficient specialized processors and CPU security extensions. While serverless computing's programming and cost model are appealing to developers, the lack of support for heterogeneous hardware in today's FaaS platforms limits practical use-cases [36]. It is not clear how to map current FaaS functions, which often intersperse computations with cloud API calls [49], to non-CPU hardware.

Nevertheless, extending the serverless paradigm to heterogeneous hardware is an active area of research. Since Kubernetes and Docker support GPU nodes, adding GPU-enabled containers into FaaS frameworks is an obvious step towards heterogeneous serverless [29, 43, 55, 73]. Another technique for heterogeneous compatibility is to provide an API-translating shim as Molecule [18] does for FPGAs and Zhao *et al.* [73] does for GPUs. Kernel-as-a-service [47] takes the approach closest to Dandelion's separation of compute and I/O by allowing users to explicitly define GPU kernels, allowing a FaaS system to mix CPU and GPU functions. We argue that, in the end, these well-engineered solutions shoehorn non-CPU execution units into a cloud that has evolved for decades around the CPU.

4 DANDELION: A NEW VISION FOR FAAS

Instead of retrofitting existing system software to meet FaaS application needs, we ask *how would we design a clean slate FaaS system software stack?* We propose *Dandelion*, a serverless computing platform that rethinks the FaaS programming model and function execution system to improve performance and energy efficiency, while maintaining secure isolation guarantees.

The key idea of Dandelion is to strictly separate compute tasks (i.e., arbitrary user computations) and I/O tasks (i.e., tasks that prepare the inputs and manage the outputs of compute tasks) in FaaS applications. Separating compute and I/O in the programming model (§4.1) and function execution model brings three key advantages. First, we can execute applications — expressed as directed acyclic graphs (DAGs) of pure compute functions (containing untrusted user code) and I/O functions (implemented by the trusted Dandelion platform) — with a more lightweight function sandbox design, without sacrificing security (§4.2). Functions can execute as lightweight threads (for performance) with hardware-enforced memory isolation and no OS interface for untrusted user code (for security). Second, Dandelion's strict separation of compute and I/O functions makes both more amenable to heterogeneous hardware acceleration to improve performance and energy efficiency (§4.3). Finally, expressing applications as DAGs of pure compute and I/O exposes dataflow to the underlying platform, enabling cluster scheduling and data movement optimizations (§4.4).

We build an initial prototype of Dandelion which we evaluate in §5. While the rest of this section presents our overarching vision for Dandelion, our current prototype focuses on demonstrating the benefits of decomposing applications into compute and I/O using a particular memory isolation mechanism (CHERI memory capabilities). Offloading functions to heterogeneous hardware and exploring dataflow-aware scheduling policies remain future work.

4.1 Programming Model

Dandelion's programming model strictly separates compute and I/O by requiring developers to express their application as a *composition* of pure compute and I/O functions. *Compute functions* contain untrusted user code and have no direct OS interface during function execution. Prohibiting system calls for compute functions avoids the risk of untrusted code exploiting vulnerabilities in the OS interface for privilege escalation and tampering with the system. On the other hand, *I/O functions* are implemented by the trusted Dandelion platform and exposed to developers as a high-level library. I/O functions enable interactions with cloud storage, databases, and other web services, as well as other functions.

While providing I/O functions to support any possible way of network communication is infeasible,[1] it also is not necessary as most web services provide HTTP interfaces. As I/O functions execute trusted code that cannot be modified by the user, they have permissions for the limited set of syscalls necessary to carry out their functionality (e.g., networking and file system calls). The main difference between generic syscalls and Dandelion's I/O API is the abstraction exposed to the application. Syscalls by design are general (e.g., sockets) and tightly integrated into the system. Our I/O library executes isolated functions performing specialized tasks (e.g., HTTP processing) allowing us to compartmentalize I/O functions and perform tailored input sanitization. Security issues may remain even with higher-level/narrower APIs but are easier to contain within single-task I/O functions as compared to the much broader Linux ABI. Dandelion avoids memory management system calls during function execution by pre-allocating an isolated memory region for each function, as we will discuss in §4.2.

Dandelion provides a simple domain-specific language for users to express applications (i.e., compositions) as DAGs that describe how data flows between functions. The domain-specific language does not generate any function code; users provide the implementation of functions as binaries or as

[1]As new hardware security primitives emerge, they may enable Dandelion to support untrusted I/O functions, enabling support for more protocols.

```
1  input item eventPattern, serverList
2  output item  totalSum
3  ephemeral set  requestSet, logSet, partialSums
4
5  requestSet = makeGetRequests(serverList)
6  logSet = for each getRequest in requestSet :
     system_HTTPGet(getRequest)
7  partialSums = for each log in logSet :
     sumOccurance(log, eventPattern)
8  totalSum = sumSet(partialSums)
```

Figure 2: Example of a composition counting occurrences of a pattern in service responses

source code to be compiled by the platform. Since compute functions cannot directly interact with the OS, Dandelion requires each function's inputs and outputs to be explicitly specified. Inputs and outputs can be *direct data*, meaning data that enters or leaves the platform attached to the function invocation request, or *ephemeral data*, which is produced by one function and consumed by another. Developers also specify the input and output data types either as items, which are contiguous arrays of bytes, or sets of items.

Figure 2 shows an example map-reduce style composition that counts the occurrences of a specific event in a set of log files. The composition uses three compute functions (`makeGetRequests`, `sumOccurance`, `sumSet`) and one I/O function (`system_HTTPGet`). We assume the user has already registered each compute function with the Dandelion platform by providing each function's source code[2] (which consists of pure computations) along with the function's number and type of input and output arguments. The composition takes two direct inputs: `eventPattern` is a string pattern identifying an event and `serverList` is list of servers to query. The composition has one direct output, `totalSum`, which will contain the number of events found. The composition also includes ephemeral variables that express dataflow between functions and are only in scope during the composition execution. Line 5 parses the input `serverList` and prepares a HTTP GET request for each server. In line 6, the composition calls the `system_HTTPGet` I/O library function to issue the GET requests. The `for each _ in _` syntax invokes parallel functions. Line 7 sums all occurrences of the event pattern for each log. Line 8 sums occurrences across all logs and stores them into the output variable `totalSum`.

To simplify porting existing applications to Dandelion's programming model and to support developing compute functions in high-level languages, such as Python whose runtime interacts extensively with the file system, Dandelion provides a custom libc library. This allows functions to

[2]Users could also upload binaries, but would need to do so for each type of hardware they want their function to be potentially run on.

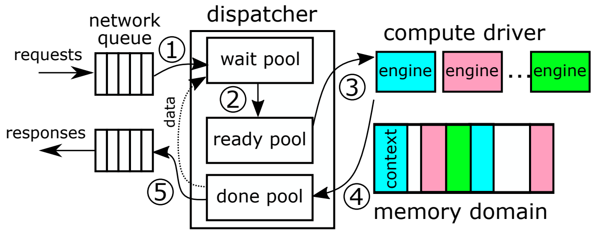

Figure 3: Dandelion function lifecycle

call traditional libc APIs, which provide a variety of memory and file management functionality. Under the hood, the custom library implements memory and file management as regular function calls that operate on the memory region pre-allocated to the function by the Dandelion platform, instead of as traditional syscalls. Dandelion's custom low-level library also keeps track of a simple structure that contains information about the memory layout and locations of the function's inputs and outputs.

For now, Dandelion expects developers to manually express their applications as compositions of pure compute functions and I/O functions. As future work, we will explore automating decomposition [22]. For example, we can use continuations [52] to split applications at I/O boundaries.

4.2 Function Execution System

We now describe how Dandelion leverages strict separation of compute and I/O to execute functions efficiently with secure isolation.

System architecture: Each worker node in a Dandelion cluster runs Dandelion's function execution system, which consists of a *dispatcher*, *memory domain managers*, and *compute drivers*. The dispatcher is the core part of a Dandelion worker, responsible for demultiplexing incoming function invocation requests for the worker, assigning sandboxes for function execution, and keeping track of the run state and data dependencies of a worker's functions. The memory domain managers and compute drivers provide the dispatcher with a high-level interface for memory and compute resource management, respectively, which the dispatcher uses to create and teardown sandboxes. A memory domain manager controls a memory region and is responsible for enforcing memory isolation between each context (i.e., subpart) of the memory domain. A compute driver is responsible for scheduling function sandboxes on a group of compute resources (i.e., compute engines), such as CPU cores.

The memory domain manager and compute driver allow us to explore and support different isolation mechanisms and hardware. For example, a memory domain manager can implement memory isolation by using page-level permissions in an MMU or more fine-grained mechanisms such as CHERI memory capabilities [68], which are available on some experimental CPU architectures [8, 67]. As an alternative to hardware isolation we could also consider software solutions such as verified WebAssembly. Pure compute functions do not need the WebAssembly System Interface, which shrinks the necessary trusted compute base by a significant margin and eliminates some of the composability problems mentioned in §3.1. Additionally we plan to leverage the higher abstraction level the memory domain manager and compute driver provide to extend the system to less conventional hardware such as GPUs, TPUs, SmartNICs or FPGAs.

Furthermore, with the narrow and clearly scoped functionality of the dispatcher, memory domain managers, and compute drivers, there is potential to formally verify these components. Rather than relying on the defense-in-depth approach to security for traditional VMs, Dandelion can increase trustworthiness by relying on a small layer of provably correct software and a few hardware primitives.

Life of a request: When a function composition invocation request arrives at a Dandelion server, the dispatcher demultiplexes the request and adds all functions that are part of the composition to the waiting pool (① in Figure 3). If the code for a function is not already in memory from a prior execution, the dispatcher will either load it from local disk (if available) or initiate fetching the code from remote storage. The dispatcher will fetch function executables compatible with one or more locally-available compute drivers. A function becomes ready (②) when all its inputs and code are available in memory. This can happen immediately on arrival if the request includes all of a function's inputs or when other functions complete and their outputs become available. To start executing a ready function, the dispatcher selects an engine type on which to run the function. It then asks a memory domain manager for a context compatible with that engine type and fills in the function's input data at a pre-specified memory address. The fully prepared context is run on an engine supplied by the corresponding compute driver. (③). To minimize the risk of side-channel attacks, engines run each function to completion. This is akin to how cloud providers like AWS minimize side channel risks by not sharing physical CPU cores between tenants [5]. When the function exits, it transitions from running to done state (④). The dispatcher cleans up the completed function's context. (⑤). Clean up involves extracting all outputs from the context and passing them on, either to other waiting functions on the worker node or as part of a response to the user. After handling all outputs, the dispatcher returns the context and engine to the memory domain manager and compute driver, respectively, to be sanitized and prepared for reuse.

4.3 Hardware Acceleration

By strictly separating compute and I/O, Dandelion makes function execution more amenable to hardware acceleration.

Accelerating I/O functions: I/O functions are only exposed to users as a library, allowing for transparent use of modern networking hardware to offload protocol processing or even entire I/O functions to SmartNICs. In addition to optimizing I/O latency and throughput, leveraging modern networking hardware can benefit the platform in several ways. Offloading frees up CPU cycles, which can be used to increase compute function throughput. Additionally, offloading can decrease total energy consumption by processing I/O requests on more energy-efficient devices. Furthermore, ensuring that I/O functions (and potentially other trusted platform code, such as the dispatcher) execute on physically separate hardware than the CPU cores running untrusted compute functions provides extra protection against attacks. This is akin to the AWS Nitro [5] approach to I/O security and performance, which offloads I/O virtualization to specialized hardware and physically separates this functionality from the software hypervisor.

Accelerating compute functions: As Dandelion compute functions are pure functions, they can be compiled and optimized for heterogeneous hardware platforms more easily than functions that interleave computation with I/O or other OS interaction. To maintain a "serverless" paradigm, Dandelion can keep the selected hardware execution platform abstracted from users. For example, Google's XLA [54] compiles Python code to execute on CPUs, GPUs, and TPUs. Dandelion users can write code for compute functions that is compatible with heterogeneous hardware compiler infrastructure to leverage CPU extensions like SIMD or transparently offload entire compute functions to accelerators like GPUs, TPUs, or FPGAs. Even if the user does not code specifically for hardware acceleration, compilers can more effectively apply optimizations on pure functions as they allow for stronger assumptions about side effects.

At first glance, short-running functions with small resource footprints may not seem like an ideal workload for hardware acceleration: loading/unloading state to a PCI-attached accelerator is slow and a single function may not make use of the numerous available compute units. However, many types of applications that stand to benefit from serverless resource management (e.g., real-time DNN inference) require heterogeneous hardware like GPUs to meet performance and energy consumption objectives. Hence, heterogeneous hardware support for compute functions can improve FaaS application latency. Furthermore, although a single function may not consume an entire accelerator, multi-tenancy support is becoming common in GPUs [44] and FPGAs [50, 51, 34]. Offloading compute functions to hardware accelerators can help improve specialized hardware's currently low average utilization in datacenters [70, 71]. Overall, heterogeneous hardware support gives greater flexibility for function scheduling, to optimize hardware utilization, energy consumption, and overall throughput.

4.4 Dataflow-Aware Scheduling

As Dandelion users develop applications as compositions of compute and I/O functions, the dataflow between functions is made explicit to the execution platform. Dandelion can leverage dataflow information to avoid or minimize data movement in several ways.

Just in time scheduling: Since function inputs are explicit, the dispatcher only schedules functions whose inputs are guaranteed to be available. This ensures that functions do not consume compute resources, which could have been used more efficiently by other functions, while waiting for inputs. Once a compute function begins executing, it does not block. Hence, compute functions can also be run to completion, which improves cache locality [11] and reduces the risk of side channel attacks by avoiding executing untrusted functions concurrently on a physical core [5].

Data locality: For compositions that pass data between functions, the cluster manager can prioritize scheduling producers and consumers on the same machine. This avoids transmitting ephemeral data over the network and enables further optimizations, like zero-copy in-memory data passing between functions [61].

Efficient distributed processing: Running all functions of a composition on a single machine is not always optimal. For example, when a composition has a high degree of parallelism, its functions can be executed concurrently across machines [38]. In such cases, the cluster manager can use information in the dataflow graph to split the composition into smaller parts in a way that minimizes the number of data items exchanged between sub-compositions.

Caching: Given the same inputs, pure functions will produce the same outputs.[3] Hence, Dandelion can cache the outputs of functions that are often invoked with the same arguments and avoid re-computation. Dandelion can also cache the outputs of I/O functions (e.g., data fetched from external storage services like S3) if the data is not expected to change and no side effects are lost.

Near storage computation: As function inputs are explicitly specified, Dandelion can identify in advance what data a composition needs to access. Compute functions can be scheduled on worker nodes close to storage nodes that hold the data or even directly on storage nodes if they have sufficient resources, thus eliminating network transfers.

[3]Randomness or time also need to be inputs to the function if they are used for computation, otherwise the function would not be pure.

5 PROOF OF CONCEPT

As a proof of concept, we prototype Dandelion's worker node function execution system. Our goal is to demonstrate Dandelion's potential to close the performance and energy efficiency gap between state-of-the-art FaaS system software and bare-metal function execution. A cluster manager for Dandelion for dataflow-aware scheduling and supporting hardware acceleration are future work.

Dandelion prototype: Our current prototype consists of 2500 lines of Rust and 740 lines of C with some inline assembly. We focused on implementing support to execute compute functions with inputs available in memory. We are actively working on adding support for I/O functions and compositions, to realize our full vision of Dandelion's function execution system.

For memory isolation, we leverage Capability Hardware Enhanced RISC Instructions (CHERI) [68] to run multiple compute functions concurrently in a single address space. CHERI is a set of CPU extensions that implements memory capabilities as an alternative to traditional pointers, adding a bounded memory range and set of permissions to each integer pointer. CHERI enforces memory bounds and permission checks for each memory access and ensures these capability bounds and permissions cannot be increased. While Dandelion can implement memory isolation with other hardware mechanisms, including traditional memory management units (MMUs), CHERI is a good fit for our use case as it enables isolation of fine-grained and arbitrarily sized memory regions.

We build and evaluate our prototype on top of Linux running on an Arm Morello [8] platform, an experimental architecture that adds CHERI support to ARMv8 CPU cores. The dispatcher in our prototype runs a simple multi-threaded HTTP service that accepts requests and prepares two minimal capabilities for each function: one for the function's code region and one for its data region. Cold requests are those for which the function's code is loaded from disk, whereas hot requests already have the code in memory.

Metrics: To evaluate the benefits of lightweight isolation we consider two metrics: end-to-end request latency and peak achievable throughput. Tail latency is an important user performance metric, while the peak throughput indicates how efficiently the provider can serve requests per machine. Higher throughput implies lower cost and energy consumption as fewer machines are needed to support a given load.

Firecracker baseline: We run the Firecracker baseline on the same Morello board as the Dandelion prototype Our HTTP frontend relays requests to functions running in MicroVMs. The functions inside MicroVMs also run a simple HTTP server that accepts requests and responds with the

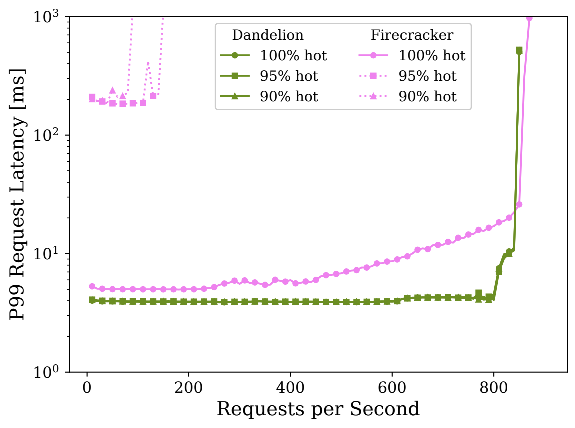

Figure 4: Dandelion and Firecracker have similar throughput for 100% hot requests, but for 95% hot Dandelion achieves 5× higher throughput.

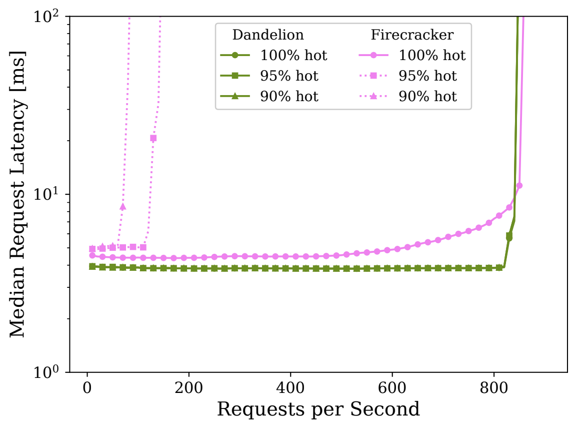

Figure 5: Firecracker median latencies for 90% and 95% hot requests degrade at the same point as tail latency, while Dandelion stays stable.

function output. For hot requests, we relay these requests to already running MicroVMs whereas for cold requests, we boot a new MicroVM using the Firecracker VMM [2] and a Linux kernel compiled with the recommended configuration.

Function workload: All functions in our experiments perform 64 bit integer matrix multiplication $M \times M^T$, where $M \in \mathbb{Z}^{128\times128}$. For both systems we send requests containing the size of the matrix and generate the input data on the machine to avoid saturating the low-bandwidth network link on the Morello board.

Experiment results: Figure 4 plots 99th percentile latency vs. throughput for Dandelion and Firecracker. We observe that Dandelion and Firecracker support similar throughput for 100% hot requests, with Dandelion showing lower latency even at high load. Dandelion also achieves 45× lower tail latency for cold starts. Thus at 95% hot requests Dandelion supports 5× higher throughput compared to Firecracker. This is because creating a function execution environment in Dandelion (i.e., allocating a compute engine and initializing a memory context) requires significantly fewer CPU cycles than booting a Firecracker MicroVM. The latency and throughput for Dandelion do not significantly change with lower rates of hot requests, because the only difference between hot and cold requests is that the binary for the function needs to be loaded from disk, which is a very cheap operation compared to booting a MicroVM.

Figure 5 plots the median response latency. In the presence of cold starts, Firecraker baseline's median latency saturates at approximately the same load as its corresponding tail latency due to the high CPU load of booting new VMs. Compute functions compete for CPU cycles to make progress and experience high queueing delays when too many VMs are trying to boot at the same time. In contrast, Dandelion's median latency stays low, similar to its tail latency.

6 CONCLUSION

Although significant effort has gone into retrofitting and optimizing legacy cloud infrastructure for FaaS, we argue that unlocking the true potential of FaaS requires revisiting assumptions in the programming model and system software design. Dandelion revisits the FaaS programming model and leverages modern hardware to provide a fast, secure, and resource-efficient serverless computing platform. The key design principle is to treat serverless functions as true functions, with a clear separation of compute and I/O. This enables a lightweight function sandbox design that maintains secure isolation while reducing tail latency by over 45× and increasing peak throughput by 5× compared to Firecracker. Dandelion's design also makes functions more amenable to hardware acceleration and dataflow-aware scheduling optimizations.

ACKNOWLEDGMENTS

The authors would like to thank Roberto Starc for his help with evaluating the Dandelion prototype. Thank you also to Gustavo Alonso, Andrea Lattuada, Lazar Cvetković, Michael Wawrzoniak, Dmitrii Ustiugov, Shweta Shinde, and Patrick Studei for fruitful discussions and feedback. Finally, we would like to thank the SoCC'23 reviewers and our shepherd, Ryan Marcus, for their helpful suggestions.

REFERENCES

[1] Mania Abdi, Samuel Ginzburg, Xiayue Charles Lin, Jose Faleiro, Gohar Irfan Chaudhry, Inigo Goiri, Ricardo Bianchini, Daniel S Berger, and Rodrigo Fonseca. 2023. Palette load balancing: locality hints for serverless functions. In *Proceedings of the Eighteenth European Conference on Computer Systems* (EuroSys '23). Association for Computing Machinery, Rome, Italy, 365–380. ISBN: 9781450394871. DOI: 10.1145/3552326.3567496.

[2] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: lightweight virtualization for serverless applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, (Feb. 2020), 419–434. ISBN: 978-1-939133-13-7. https://www.usenix.org/conference/nsdi20/presentation/agache.

[3] Amazon Web Services. 2023. AWS step functions: visual workflows for distributed applications. Amazon Web Services. Retrieved Sept. 4, 2023 from https://aws.amazon.com/step-functions/.

[4] Amazon Web Services. 2023. Security overview of AWS Lambda: AWS whitepaper. Amazon Web Services. Retrieved June 1, 2023 from https://aws.amazon.com/lambda/security-overview-of-aws-lambda/.

[5] Amazon Web Services. 2022. The security design of the AWS Nitro system. Amazon Web Services. (Nov. 18, 2022). Retrieved June 1, 2023 from https://docs.aws.amazon.com/whitepapers/latest/security-design-of-aws-nitro-system/security-design-of-aws-nitro-system.html.

[6] Android Developers. 2023. Overview of memory management. Android Developers. (May 9, 2023). Retrieved Sept. 4, 2023 from https://developer.android.com/topic/performance/memory-overview.

[7] Anjali, Tyler Caraza-Harter, and Michael M. Swift. 2020. Blending containers and virtual machines: a study of firecracker and gvisor. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (VEE '20). Association for Computing Machinery, Lausanne, Switzerland, 101–113. ISBN: 9781450375542. DOI: 10.1145/3381052.3381315.

[8] Arm Ltd. 2023. Arm morello program. Arm Ltd. Retrieved June 1, 2023 from https://www.arm.com/architecture/cpu/morello.

[9] Jeff Barr. 2022. Accelerate your lambda functions with Lambda SnapStart. Amazon Web Services. (Dec. 9, 2022). Retrieved June 1, 2023 from https://aws.amazon.com/blogs/aws/new-accelerate-your-lambda-functions-with-lambda-snapstart/.

[10] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. 2012. Dune: safe user-level access to privileged CPU features. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. USENIX Association, Hollywood, CA, (Oct. 2012), 335–348. ISBN: 978-1-931971-96-6. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/belay.

[11] 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 *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO, (Oct. 2014), 49–65. ISBN: 978-1-931971-16-4. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/belay.

[12] Zach Bloom. 2018. Cloud computing without containers. CloudFlare. (Nov. 9, 2018). Retrieved Sept. 4, 2023 from https://blog.cloudflare.com/cloud-computing-without-containers/.

[13] Marc Brooker, Adrian Costin Catangiu, Mike Danilov, Alexander Graf, Colm MacCarthaigh, and Andrei Sandu. 2021. Restoring uniqueness in microvm snapshots. (2021). arXiv: 2102.12892 [cs.CR]. DOI: 10.48550/arXiv.2102.12892.

[14] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. 2020. Seuss: skip redundant paths to make serverless fast. In *Proceedings of the Fifteenth European Conference on Computer Systems* (EuroSys '20) Article 32. Association for Computing Machinery, Heraklion, Greece, 15 pages. ISBN: 9781450368827. DOI: 10.1145/3342195.3392698.

[15] Paul Castro, Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. 2019. The rise of serverless computing. *Commun. ACM*, 62, 12, (Nov. 2019), 44–54. DOI: 10.1145/3368454.

[16] Datadog. 2021. The state of serverless (2021). Datadog. (May 1, 2021). Retrieved June 1, 2023 from https://www.datadoghq.com/state-of-serverless-2021/.

[17] Yuhan Deng, Angela Montemayor, Amit Levy, and Keith Winstein. 2022. Computation-centric networking. In *Proceedings of the 21st ACM Workshop on Hot Topics in Networks* (HotNets '22). Association for Computing Machinery, Austin, Texas, 167–173. ISBN: 9781450398992. DOI: 10.1145/3563766.3564106.

[18] Dong Du, Qingyuan Liu, Xueqiang Jiang, Yubin Xia, Binyu Zang, and Haibo Chen. 2022. Serverless computing on heterogeneous computers. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS '22). Association for Computing Machinery, Lausanne, Switzerland, 797–813. ISBN: 9781450392051. DOI: 10.1145/3503222.3507732.

[19] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. 2020. Catalyzer: sub-millisecond startup for serverless computing with initialization-less booting. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS '20). Association for Computing Machinery, Lausanne, Switzerland, 467–481. ISBN: 9781450371025. DOI: 10.1145/3373376.3378512.

[20] Simon Eismann, Joel Scheuner, Erwin van Eyk, Maximilian Schwinger, Johannes Grohmann, Nikolas Herbst, Cristina L. Abad, and Alexandru Iosup. 2022. The state of serverless applications: collection, characterization, and community consensus. *IEEE Transactions on Software Engineering*, 48, 10, 4152–4166. DOI: 10.1109/TSE.2021.3113940.

[21] Simon Eismann, Joel Scheuner, Erwin van Eyk, Maximilian Schwinger, Johannes Grohmann, Nikolas Herbst, Cristina L. Abad, and Alexandru Iosup. 2021. Serverless applications: why, when, and how? *IEEE Software*, 38, 1, 32–39. DOI: 10.1109/MS.2020.3023302.

[22] Sanjay Ghemawat, Robert Grandl, Srdjan Petrovic, Michael Whittaker, Parveen Patel, Ivan Posva, and Amin Vahdat. 2023. Towards modern development of cloud applications. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems* (HOTOS '23). Association for Computing Machinery, Providence, RI, USA, 110–117. ISBN: 9798400701955. DOI: 10.1145/3593856.3595909.

[23] Cristiano Giuffrida, Anton Kuijsten, and Andrew S. Tanenbaum. 2012. Enhanced operating system security through efficient and fine-grained address space randomization. In *21st USENIX Security Symposium (USENIX Security 12)*. USENIX Association, Bellevue, WA, (Aug. 2012), 475–490. ISBN: 978-931971-95-9. https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/giuffrida.

[24] gVisor. 2023. What is gvisor? gVisor. Retrieved Sept. 4, 2023 from https://gvisor.dev/docs/.

[25] Joseph M. Hellerstein, Jose M. Faleiro, Joseph Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang

Wu. 2019. Serverless Computing: One Step Forward, Two Steps Back. In *Proceedings of the 9th Biennial Conference on Innovative Data Systems Research (CIDR)*. (Jan. 2019). http://cidrdb.org/cidr2019/papers/p119-hellerstein-cidr19.pdf.

[26] Evan Johnson, Evan Laufer, Zijie Zhao, Dan Gohman, Shravan Narayan, Stefan Savage, Deian Stefan, and Fraser Brown. 2023. WaVe: a verifiably secure WebAssembly sandboxing runtime. In *2023 IEEE Symposium on Security and Privacy (SP)*, 2940–2955. DOI: 10.1109/SP46215.2023.10179357.

[27] Evan Johnson, David Thien, Yousef Alhessi, Shravan Narayan, Fraser Brown, Sorin Lerner, Tyler McMullen, Stefan Savage, and Deian Stefan. 2021. Trust but verify: SFI safety for native-compiled Wasm. In *Network and Distributed System Security Symposium (NDSS)*. Internet Society, (Feb. 2021).

[28] Kostis Kaffes, Neeraja J. Yadwadkar, and Christos Kozyrakis. 2019. Centralized core-granular scheduling for serverless functions. In *Proceedings of the ACM Symposium on Cloud Computing* (SoCC '19). Association for Computing Machinery, Santa Cruz, CA, USA, 158–164. ISBN: 9781450369732. DOI: 10.1145/3357223.3362709.

[29] Jaewook Kim, Tae Joon Jun, Daeyoun Kang, Dohyeun Kim, and Daeyoung Kim. 2018. GPU enabled serverless computing framework. In *2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, 533–540. DOI: 10.1109/PDP2018.2018.00090.

[30] Avi Kivity, Dor Laor, Glauber Costa, Pekka Enberg, Nadav Har'El, Don Marti, and Vlad Zolotarov. 2014. OSv—Optimizing the operating system for virtual machines. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. USENIX Association, Philadelphia, PA, (June 2014), 61–72. ISBN: 978-1-931971-10-2. https://www.usenix.org/conference/atc14/technical-sessions/presentation/kivity.

[31] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. 2018. Pocket: elastic ephemeral storage for serverless analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, (Oct. 2018), 427–444. ISBN: 978-1-939133-08-3. https://www.usenix.org/conference/osdi18/presentation/klimovic.

[32] Knative. 2023. Knative serverless containers. Knative. Retrieved Sept. 4, 2023 from https://knative.dev/docs/.

[33] Marcin Kolny. 2023. Scaling up the Prime Video audio/video monitoring service and reducing costs by 90%. Amazon Web Services. (Mar. 22, 2023). Retrieved June 1, 2023 from https://www.primevideotech.com/video-streaming/scaling-up-the-prime-video-audio-video-monitoring-service-and-reducing-costs-by-90.

[34] Dario Korolija, Timothy Roscoe, and Gustavo Alonso. 2020. Do OS abstractions make sense on FPGAs? In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, (Nov. 2020), 991–1010. ISBN: 978-1-939133-19-9. https://www.usenix.org/conference/osdi20/presentation/roscoe.

[35] Simon Kuenzer, Vlad-Andrei Bădoiu, Hugo Lefeuvre, Sharan Santhanam, Alexander Jung, Gaulthier Gain, Cyril Soldani, Costin Lupu, Ştefan Teodorescu, Costi Răducanu, Cristian Banu, Laurent Mathy, Răzvan Deaconescu, Costin Raiciu, and Felipe Huici. 2021. Unikraft: fast, specialized unikernels the easy way. In *Proceedings of the Sixteenth European Conference on Computer Systems* (EuroSys '21). Association for Computing Machinery, Online Event, United Kingdom, 376–394. ISBN: 9781450383349. DOI: 10.1145/3447786.3456248.

[36] Xiayue Charles Lin, Joseph E. Gonzalez, and Joseph M. Hellerstein. 2020. Serverless boom or bust? an analysis of economic incentives. In *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 20)*. USENIX Association, (July 2020). https://www.usenix.org/conference/hotcloud20/presentation/lin.

[37] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. 2013. Unikernels: library operating systems for the cloud. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS '13). Association for Computing Machinery, Houston, Texas, USA, 461–472. ISBN: 9781450318709. DOI: 10.1145/2451116.2451167.

[38] Ashraf Mahgoub, Karthick Shankar, Subrata Mitra, Ana Klimovic, Somali Chaterji, and Saurabh Bagchi. 2021. SONIC: application-aware data passing for chained serverless applications. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, (July 2021), 285–301. ISBN: 978-1-939133-23-6. https://www.usenix.org/conference/atc21/presentation/mahgoub.

[39] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. 2017. My vm is lighter (and safer) than your container. In *Proceedings of the 26th Symposium on Operating Systems Principles* (SOSP '17). Association for Computing Machinery, Shanghai, China, 218–233. ISBN: 9781450350853. DOI: 10.1145/3132747.3132763.

[40] Holly Mesrobian and Marc Brooker. 2019. AWS re:Invent: a serverless journey: AWS Lambda under the hood. Amazon Web Services. (Dec. 9, 2019). Retrieved June 1, 2023 from https://www.youtube.com/watch?v=xmacMfbrG28.

[41] Microsoft Azure. 2023. What are durable functions? Microsoft Azure. Retrieved June 1, 2023 from https://learn.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview.

[42] Ingo Müller, Renato MarroquıHEREHEREHEREn, and Gustavo Alonso. 2020. Lambada: interactive data analytics on cold data using serverless cloud infrastructure. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (SIGMOD '20). Association for Computing Machinery, Portland, OR, USA, 115–130. ISBN: 9781450367356. DOI: 10.1145/3318464.3389758.

[43] Diana M. Naranjo, Sebastián Risco, Carlos de Alfonso, Alfonso Pérez, Ignacio Blanquer, and Germán Moltó. 2020. Accelerated serverless computing based on GPU virtualization. *Journal of Parallel and Distributed Computing*, 139, 32–42. DOI: https://doi.org/10.1016/j.jpdc.2020.01.004.

[44] NVIDIA Corporation. 2023. NVIDIA Corporation. Retrieved Sept. 4, 2023 from https://www.nvidia.com/en-us/technologies/multi-instance-gpu/.

[45] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2018. SOCK: rapid task provisioning with serverless-optimized containers. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference* (USENIX ATC '18). USENIX Association, Boston, MA, USA, 57–69. ISBN: 9781931971447. https://www.usenix.org/conference/atc18/presentation/oakes.

[46] OpenFaas. 2023. Openfaas: serverless functions made simple. OpenFaas Ltd. Retrieved Sept. 4, 2023 from https://www.openfaas.com/.

[47] Nathan Pemberton, Anton Zabreyko, Zhoujie Ding, Randy Katz, and Joseph Gonzalez. 2022. Kernel-as-a-service: a serverless interface to GPUs. (2022). arXiv: 2212.08146 `[cs.DC]`. DOI: 10.48550/arXiv.2212.08146.

[48] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. 2019. Shuffling, fast and slow: scalable analytics on serverless infrastructure. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation* (NSDI'19). USENIX Association, Boston, MA, USA, 193–206. ISBN: 9781931971492. https://www.usenix.org/conference/nsdi19/presentation/pu.

[49] Giuseppe Raffa, Jorge Blasco Alis, Dan O'Keeffe, and Santanu Kumar Dash. 2023. AWSomePy: a dataset and characterization of serverless applications. In *Proceedings of the 1st Workshop on SErverless Systems, Applications and MEthodologies* (SESAME '23). Association for Computing Machinery, Rome, Italy, 50–56. ISBN: 9798400701856. DOI: 10.1145/3592533.3592811.

[50] Francesco Restuccia, Andres Meza, and Ryan Kastner. 2021. Aker: a design and verification framework for safe and secure soc access control. In *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE Press, Munich, Germany, 1–9. DOI: 10.1109/ICCAD51958.2021.9643538.

[51] Francesco Restuccia, Andres Meza, Ryan Kastner, and Jason Oberg. 2023. A framework for design, verification, and management of SoC access control systems. *IEEE Transactions on Computers*, 72, 2, 386–400. DOI: 10.1109/TC.2022.3209923.

[52] John C. Reynolds. 1993. The discoveries of continuations. *Lisp Symb. Comput.*, 6, 3–4, (Nov. 1993), 233–248. DOI: 10.1007/BF01019459.

[53] Francisco Romero, Gohar Irfan Chaudhry, Íñigo Goiri, Pragna Gopa, Paul Batum, Neeraja J. Yadwadkar, Rodrigo Fonseca, Christos Kozyrakis, and Ricardo Bianchini. 2021. Faa$T: a transparent auto-scaling cache for serverless applications. In *Proceedings of the ACM Symposium on Cloud Computing* (SoCC '21). Association for Computing Machinery, Seattle, WA, USA, 122–137. ISBN: 9781450386388. DOI: 10.1145/3472883.3486974.

[54] Amit Sabne. 2020. XLA : compiling machine learning for peak performance. (2020). https://research.google/pubs/pub50530/.

[55] Klaus Satzke, Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Andre Beck, Paarijaat Aditya, Manohar Vanga, and Volker Hilt. 2021. Efficient GPU sharing for serverless workflows. In *Proceedings of the 1st Workshop on High Performance Serverless Computing* (HiPS '21). Association for Computing Machinery, Virtual Event, Sweden, 17–24. ISBN: 9781450383882. DOI: 10.1145/3452413.3464785.

[56] Divyanshu Saxena, Tao Ji, Arjun Singhvi, Junaid Khalid, and Aditya Akella. 2022. Memory deduplication for serverless computing with Medes. In *Proceedings of the Seventeenth European Conference on Computer Systems* (EuroSys '22). Association for Computing Machinery, Rennes, France, 714–729. ISBN: 9781450391627. DOI: 10.1145/3492321.3524272.

[57] David Schall, Artemiy Margaritov, Dmitrii Ustiugov, Andreas Sandberg, and Boris Grot. 2022. Lukewarm serverless functions: characterization and optimization. In *Proceedings of the 49th Annual International Symposium on Computer Architecture* (ISCA '22). Association for Computing Machinery, New York, New York, 757–770. ISBN: 9781450386104. DOI: 10.1145/3470496.3527390.

[58] Johann Schleier-Smith, Vikram Sreekanti, Anurag Khandelwal, Joao Carreira, Neeraja J. Yadwadkar, Raluca Ada Popa, Joseph E. Gonzalez, Ion Stoica, and David A. Patterson. 2021. What serverless computing is and should become: the next phase of cloud computing. *Commun. ACM*, 64, 5, (Apr. 2021), 76–84. DOI: 10.1145/3406011.

[59] Mohammad Shahrad, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the wild: characterizing and optimizing the serverless workload at a large cloud provider. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference* (USENIX ATC'20) Article 14. USENIX Association, USA, 14 pages. ISBN: 978-1-939133-14-4. https://www.usenix.org/conference/atc20/presentation/shahrad.

[60] Zhiming Shen, Zhen Sun, Gur-Eyal Sela, Eugene Bagdasaryan, Christina Delimitrou, Robbert Van Renesse, and Hakim Weatherspoon. 2019. X-containers: breaking down barriers to improve performance and isolation of cloud-native containers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS '19). Association for Computing Machinery, Providence, RI, USA, 121–135. ISBN: 9781450362405. DOI: 10.1145/3297858.3304016.

[61] Simon Shillaker and Peter Pietzuch. 2020. FAASM: lightweight isolation for efficient stateful serverless computing. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference* (USENIX ATC'20) Article 28. USENIX Association, USA, 15 pages. ISBN: 978-1-939133-14-4. https://www.usenix.org/conference/atc20/presentation/shillaker.

[62] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. 2021. Benchmarking, analysis, and optimization of serverless function snapshots. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS '21). Association for Computing Machinery, Virtual, USA, 559–572. ISBN: 9781450383172. DOI: 10.1145/3445814.3446714.

[63] Ao Wang, Shuai Chang, Huangshi Tian, Hongqi Wang, Haoran Yang, Huiba Li, Rui Du, and Yue Cheng. 2021. FaaSNet: scalable and fast provisioning of custom serverless container runtimes at Alibaba cloud function compute. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, (July 2021), 443–457. ISBN: 978-1-939133-23-6. https://www.usenix.org/conference/atc21/presentation/wang-ao.

[64] Kai-Ting Amy Wang, Rayson Ho, and Peng Wu. 2019. Replayable execution optimized for page sharing for a managed runtime environment. In *Proceedings of the Fourteenth EuroSys Conference 2019* (EuroSys '19) Article 39. Association for Computing Machinery, Dresden, Germany, 16 pages. ISBN: 9781450362818. DOI: 10.1145/3302424.3303978.

[65] Sutao Wang. 2021. *Thin Serverless Functions with GraalVM Native Image*. Master's thesis. ETH Zurich, (Apr. 22, 2021). DOI: 10.3929/ethz-b-000480335.

[66] Nicholas C. Wanninger, Joshua J. Bowden, Kirtankumar Shetty, Ayush Garg, and Kyle C. Hale. 2022. Isolating functions at the hardware limit with virtines. In *Proceedings of the Seventeenth European Conference on Computer Systems* (EuroSys '22). Association for Computing Machinery, Rennes, France, 644–662. ISBN: 9781450391627. DOI: 10.1145/3492321.3519553.

[67] Robert N. M. Watson. 2023. CHERI RISC-V Project Page. University of Cambridge. Retrieved Sept. 4, 2023 from https://www.cl.cam.ac.uk/research/security/ctsrd/cheri/cheri-risc-v.html.

[68] Robert N.M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, Steven J. Murdoch, Robert Norton, Michael Roe, Stacey Son, and Munraj Vadera. 2015. CHERI: a hybrid capability-system architecture for scalable software compartmentalization. In *2015 IEEE Symposium on Security and Privacy*, 20–37. DOI: 10.1109/SP.2015.9.

[69] Michal Wawrzoniak, Ingo Müller, Gustavo Alonso, and Rodrigo Bruno. 2021. Boxer: data analytics on network-enabled serverless platforms. In *11th Conference on Innovative Data Systems Research, CIDR 2021, Virtual Event, January 11-15, 2021, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2021/papers/cidr2021%5C_paper12.pdf.

[70] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. 2022. MLaaS in the wild: workload analysis and scheduling in Large-Scale heterogeneous GPU clusters. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association,

Renton, WA, (Apr. 2022), 945–960. ISBN: 978-1-939133-27-4. https://www.usenix.org/conference/nsdi22/presentation/weng.

[71] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. 2020. AntMan: dynamic scaling on GPU clusters for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, (Nov. 2020), 533–548. ISBN: 978-1-939133-19-9. https://www.usenix.org/conference/osdi20/presentation/xiao.

[72] Minchen Yu, Tingjia Cao, Wei Wang, and Ruichuan Chen. 2023. Following the data, not the function: rethinking function orchestration in serverless computing. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, (Apr. 2023), 1489–1504. ISBN: 978-1-939133-33-5. https://www.usenix.org/conference/nsdi23/presentation/yu.

[73] Ming Zhao, Kritshekhar Jha, and Sungho Hong. 2023. GPU-enabled function-as-a-service for machine learning inference. (2023). arXiv: 2303.05601 [cs.DC]. DOI: 10.48550/arXiv.2303.05601.