MXFaaS: Resource Sharing in Serverless Environments for Parallelism and Efficiency

Jovan Stojkovic
University of Illinois at Urbana-Champaign
jovans2@illinois.edu

Tianyin Xu
University of Illinois at Urbana-Champaign
tyxu@illinois.edu

Hubertus Franke
IBM Research
frankeh@us.ibm.com

Josep Torrellas
University of Illinois at Urbana-Champaign
torrella@illinois.edu

ABSTRACT

Although serverless computing is a popular paradigm, current serverless environments have high overheads. Recently, it has been shown that serverless workloads frequently exhibit bursts of invocations of the same function. Such pattern is not handled well in current platforms. Supporting it efficiently can speed-up serverless execution substantially.

In this paper, we target this dominant pattern with a new serverless platform design named *MXFaaS*. MXFaaS improves function performance by efficiently *multiplexing* (i.e., sharing) processor cycles, I/O bandwidth, and memory/processor state between concurrently executing invocations of the same function. MXFaaS introduces a new container abstraction called *MXContainer*. To enable efficient use of processor cycles, an MXContainer carefully helps schedule same-function invocations for minimal response time. To enable efficient use of I/O bandwidth, an MXContainer coalesces remote storage accesses and remote function calls from same-function invocations. Finally, to enable efficient use of memory/processor state, an MXContainer first initializes the state of its container and only later, on demand, spawns a process per function invocation, so that all invocations can share unmodified memory state and hence minimize memory footprint.

We implement MXFaaS in two serverless platforms and run diverse serverless benchmarks. With MXFaaS, serverless environments are much more efficient. Compared to a state-of-the-art serverless environment, MXFaaS on average speeds-up execution by 5.2×, reduces P99 tail latency by 7.4×, and improves throughput by 4.8×. In addition, it reduces the average memory usage by 3.4×.

CCS CONCEPTS

• **Computer systems organization** → **Cloud computing**; **Distributed architectures**;

KEYWORDS

Serverless computing, cloud computing, resource management

ISCA '23, June 17–21, 2023, Orlando, FL, USA

ACM ISBN 979-8-4007-0095-8/23/06…$15.00
https://doi.org/10.1145/3579371.3589069

ACM Reference Format:
Jovan Stojkovic, Tianyin Xu, Hubertus Franke, and Josep Torrellas. 2023. MXFaaS: Resource Sharing in Serverless Environments for Parallelism and Efficiency. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA '23), June 17–21, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3579371.3589069

1 INTRODUCTION

Serverless computing is a popular cloud computing paradigm. Users upload their application and the cloud provider secures all the libraries, runtime environment, and system services needed to run it. The basic unit of execution is a function, which runs in an ephemeral, stateless Container or micro virtual machine (VM) created and scheduled on demand in an event-driven manner. Applications are then composed of multiple functions that communicate with each other. In this environment, applications have the potential to attain high resource utilization, scale easily, and can be billed in a fine-grained manner. Today, serverless cloud services are offered by all major cloud providers [2, 25, 30, 50] and are widely used to construct applications in e-commerce [42, 78], image and video processing [3, 22], machine learning model training and inference [32, 65, 79], and many other domains [21].

Despite these attractive benefits, current serverless workloads suffer multiple overheads. The most important ones include lengthy startup latency [14, 18, 46, 66, 70, 75], large memory footprints that limit how many containers can be executing concurrently [24], and frequent execution stalls due to RPC/HTTP invocations to call functions or to access remote storage [31, 36, 40, 45, 49, 64, 77]. Existing techniques address some of these shortcomings, but a lot of improvement is still needed. For example, to reduce startup overhead, some systems keep the state of an idle container in memory for a long time [24, 51, 70], aggravating memory limitations.

Recent measurements have revealed that serverless workloads frequently exhibit bursts of invocations of the same function [70, 76], either from different end-users or from a single one. Different end-users can create bursts of invocations to popular functions, triggered by certain events. A single end-user may issue thousands of invocations of the same function in seconds—e.g., in serverless video processing systems like ExCamera and Sprocket, to parallelize real-time video encoding [3, 22]. In response to either case, current serverless platforms spawn and execute many containers concurrently.

An analysis of state-of-the-art platforms shows how inefficiently this pattern is supported. First, the execution of a function is most of the time blocked on synchronous wait operations, and cores either

remain idle or frequently context switch between containers of different functions. Hence, response times easily degrade. Second, concurrent execution of multiple invocations of the same function causes repeated I/O accesses to the same or similar data, and calls to the same remote functions. The result is inefficient I/O bandwidth use. Finally, the different invocations of the same function largely bring the same state to memory. If the system does not allow memory sharing between invocations, the replicated state consumes substantial memory, inhibiting the execution of other containers and increasing their startup overhead.

To improve performance under this typical behavior, this paper introduces a new serverless platform design named *MXFaaS* or *Multiplexed FaaS*. MXFaaS improves performance by efficiently multiplexing (i.e., sharing) processor cycles, I/O bandwidth, and memory/processor state between concurrently-executing invocations of the same function.

MXFaaS introduces *MXContainer*, a new container abstraction that can concurrently execute multiple invocations of a single function and owns a set of cores. An MXContainer has a *Dispatcher* process and multiple *Handler* processes. The dispatcher initializes the container in the first function invocation. At every function invocation, the dispatcher forks a handler to serve the request. MXFaaS introduces three techniques, which address each of the three aforementioned inefficiencies.

First, to enable efficient use of processor cycles, the dispatcher in an MXContainer carefully suspends and resumes its handlers. Its aim is to ensure that, at any time, the OS can schedule the *oldest N ready-to-execute invocations* of the function—where *N* is the number of cores owned by the MXContainer. The resulting execution minimizes function response time.

Second, to enable efficient use of I/O bandwidth, the dispatcher coalesces remote storage accesses and remote function calls from multiple invocations of the same function. The coalesced storage requests can be for a single key or for a vector of them; the coalesced function calls are for the same function. The result is lower network demands and reduced pressure on storage and processors.

Finally, to enable efficient use of memory/processor state, the dispatcher first initializes the state of the container and only later, on demand, spawns a handler process per function invocation. With this design, all invocations share the unmodified initialization state (which may comprise Mbytes of memory), reducing overall memory footprint and allowing more instances of functions to reside in memory at a time. Further, by executing these multiple invocations of the same function on the owned cores, one also reuses the cache and branch predictor state.

There are some prior schemes that enable some reuse of memory state across concurrent invocations of the same function [1, 18, 31, 52]—but not to the extent of our proposal. The details are in Section 10. No prior scheme combines the use of CPU cycles or I/O bandwidth across concurrent invocations of the same function.

We implement MXFaaS in the Apache OpenWhisk [5] and KNative [37] platforms. MXFaaS does not require any hardware or operating system (OS) support, or changes to user functions. We evaluate MXFaaS with a diverse set of serverless benchmarks and show that MXFaaS is very effective. Compared to a state-of-the-art serverless baseline [31], MXFaaS on average speeds-up execution by 5.2×, reduces the P99 tail latency by 7.4×, and improves throughput by 4.8×. In addition, it reduces the average memory usage by 3.4×. Finally, MXFaaS outperforms an ideal scheme that predicts which containers will be needed next and proactively warms them up, by an average of 2.1× (or 2.9× for high load).

This paper makes the following contributions:

- A characterization of state-of-the-art serverless systems.
- The MXContainer abstraction.
- The MXFaaS serverless platform that enables efficient use of processor cycles, I/O bandwidth, and memory state.
- An implementation and evaluation of MXFaaS.

2 BACKGROUND: FAAS PLATFORM

Figure 1 illustrates a typical architecture of a serverless platform, such as OpenWhisk, KNative, OpenFaaS, or OpenLambda [5, 19, 20, 29, 37, 41, 53, 75]. A platform consists of centralized control modules (e.g., the *frontend* and the *load balancer*) that accept function invocations and distribute them to the nodes. In each node, there is an *invoker* module that is responsible for the execution of the function.

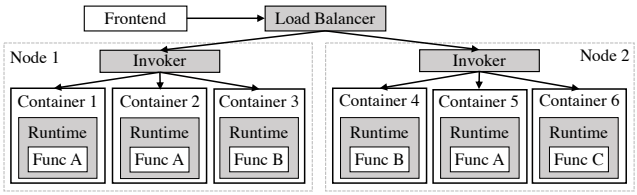


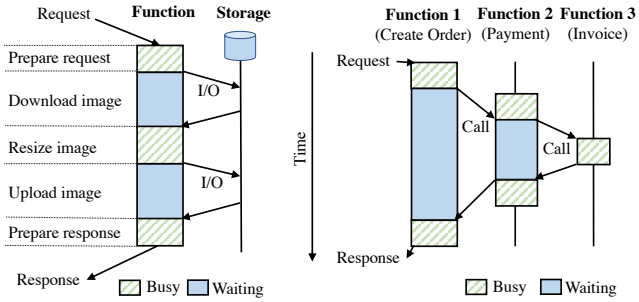
Figure 1: Overview of existing serverless platforms.

To execute a user-provided function, an invoker encapsulates the function code together with an execution runtime in a container, and spawns the runtime process [56]. The user function is executed inside the address space of the runtime process [8]. The runtime process first executes initialization code to set up network connections and initialize global variables. Then, it invokes the requested function. The runtime only services one request (i.e., one function invocation) at a time. To execute multiple invocations of the same function concurrently, production-level platforms need to spawn multiple containers.

When a function invocation completes, serverless platforms keep the container in memory in warm state for a certain period of time [24, 51, 70]. If, during this period, another request for the same function is received, it is executed by the runtime process in the warm container. However, with this approach, the global state of the runtime process is sequentially shared across function invocations. Such sharing may lead to both security and correctness issues [8].

3 CHARACTERIZING FAAS ENVIRONMENTS

To understand the performance of serverless environments, this section analyzes real-world serverless workloads and open-source serverless benchmarks running on OpenWhisk [5]. For the former, we use production traces from Azure's serverless functions [11, 64] and Alibaba's microservices [48]; for the latter, we use functions from FunctionBench [35] and applications from TrainTicket [68, 83]. We give more details of all these workloads in Section 7.



(a) Anti-pattern 1: Synchronous I/O (b) Anti-pattern 2: Functions calling functions

Figure 2: Inefficient function patterns: (a) synchronous I/O within a function, and (b) functions calling functions.

3.1 Inefficient Patterns

We observe a few inefficiencies in function implementation, application orchestration, and resource provisioning.

3.1.1 Synchronous I/O within a Function. Serverless functions rely on remote storage to maintain state. Functions are often of millisecond scale [70], and storage I/O can easily dominate function execution time. Therefore, synchronous I/O in function code, as shown in Figure 2a, is strongly discouraged as an anti-pattern [7]. Since synchronous I/O is often needed due to data dependencies, the recommendation is to split a function into multiple smaller functions and perform the I/O in between two functions. However, it may not always be feasible to prevent synchronous I/O in functions. Moreover, developers often fail to use disciplined coding practices.

3.1.2 Functions Calling Other Functions. To orchestrate serverless applications, it is common to let a function call other dependent functions—which resembles procedure calls in traditional programming. Such practice is also an anti-pattern because such RPCs result in a compound effect of synchronous waits, as shown in Figure 2b. Although this anti-pattern is also strongly discouraged [6], we find that it is prevalent in existing serverless applications. One reason is that many serverless applications originate from traditional microservice applications that use RPCs to orchestrate applications.

3.1.3 Minimizing Function Startup Time. To minimize the startup overhead of function invocations, a number of optimizations have been developed. One approach is to keep an idle container in memory for long, so its process and state can be reused when a subsequent invocation of the same function arrives [24, 51, 70]. This optimization tends to increase the memory footprint—potentially preventing other functions from executing concurrently due to lack of memory. In addition, reusing the process in a container breaks the isolation expected of containers: the state generated by one function invocation is visible to the next invocation [8].

Another approach is to keep container snapshots in disk and preload one when a request for the corresponding function arrives [18, 75]. This approach and the previous one speed-up the startup of individual requests, rather than targeting many concurrent requests in a burst. As a result, on a burst, they consume substantial memory or create substantial disk traffic.

Other schemes such as SAND [1] and Faastlane [40] minimize startup time by creating a single container for all the different functions of an application. However, this design makes it harder to efficiently manage the hardware resources per container and scale the number of containers, because the different functions in a container may have very different hardware requirements and software dependencies. For scalability and resource management efficiency, it is best to keep different functions in different containers.

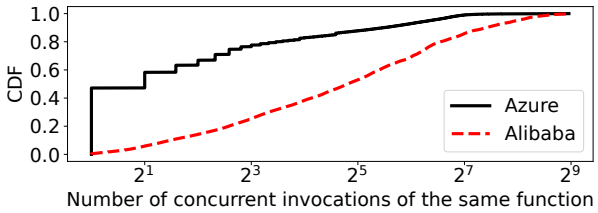


Figure 3: CDF of the number of concurrent invocations of the same function in Azure and Alibaba FaaS traces.

3.2 Workload and Execution Characteristics

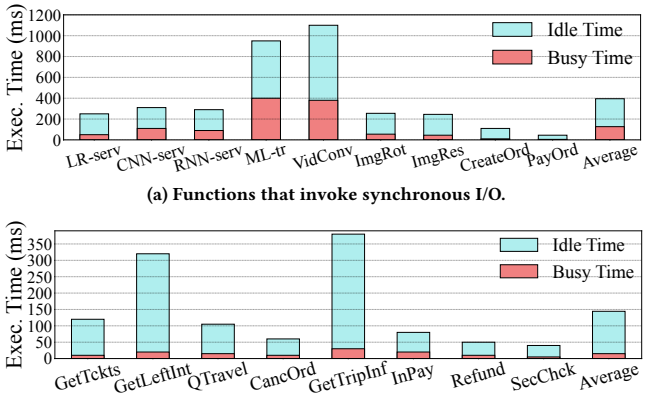
3.2.1 Invocations of the Same Function are Bursty. It is known that serverless workloads exhibit bursts of invocations of the same function [70, 76]. Figure 3 shows the distribution of the number of concurrent invocations of the same function in real-world workloads, based on production FaaS traces from Azure [11] and microservice traces from Alibaba [48]. The figure shows the CDF distribution. In Alibaba, 50% of the invocations of a function are in bursts of 32 or more concurrent invocations of the function. Azure traces are less skewed, but still, 20% of the invocations of a function are in bursts of 8 or more concurrent invocations. This data is a result of the goal of the serverless computing model to promote autoscaling and elasticity.

3.2.2 Idle Time Dominates Function Execution. We take serverless functions from FunctionBench [35] and serverless applications from TrainTicket [68], and measure the idle time during the execution of each function. We find that the two stall patterns of Section 3.1.1 and Section 3.1.2 are prevalent. All the 47 functions in the two suites exhibit one of the two patterns. We also inspect other serverless benchmarks [9, 15, 45, 80] and observe the same patterns.

Figure 4 shows the busy and idle time of a representative set of these functions. Figure 4a shows functions that invoke synchronous I/O. On average, 68% of the execution time of these functions is taken by idle time. It can be shown that all the 14 functions in FunctionBench issue synchronous I/O requests, following the procedure of Figure 2a, where the code first downloads data from remote storage, then processes it, and finally uploads the results to the storage.

Figure 4b shows functions that call other functions. On average, 90% of the execution time of these functions is idle time. In TrainTicket, it is common to have a calling pattern as in Figure 2b. Of the 33 functions in TrainTicket, 13 have RPCs and the remaining ones issue synchronous I/O (e.g., CreateOrd and PayOrd in Figure 4a). The 13 functions that use RPCs issue on average 4.8 RPCs. As the leaf functions use synchronous I/O, the inefficiency propagates along the call chains (Figure 2b).

3.2.3 There Is Substantial State Replication in Memory. When multiple invocations of the same function execute concurrently, they



(a) Functions that invoke synchronous I/O.

(b) Functions that call other functions.

Figure 4: Busy and idle time of a representative set of functions from FunctionBench and TrainTicket.

frequently access the same data and instructions. Unless a deliberate effort is made to ensure that the invocations share pages, a lot of data will be replicated in memory. The resulting large memory footprint will limit the number of containers that can reside in memory at a time and hurt throughput.

To understand the extent of the problem, we measure the memory footprint of each individual function in Figure 4a and break it into the three categories shown in Figure 5: *LibLd* is the footprint of the shared libraries; *Init* is the footprint of read-only data that is function-specific and independent of individual invocations of the function; and *Handler* is the footprint of the per-invocation private data. The bars are normalized to 1 and, on top of each, we show the total footprint in Mbytes. On average, *LibLd*, *Init*, and *Handler* account for 66%, 24%, and 10% of the total footprint, respectively.

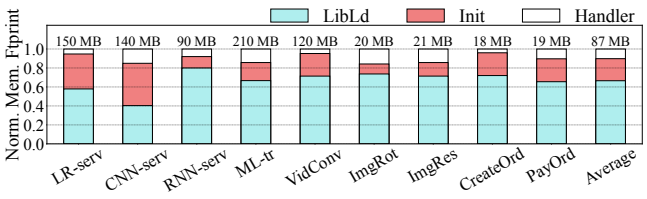


Figure 5: Breakdown of the normalized memory footprint.

Under different FaaS schemes, concurrent invocations of the same function can share different parts of the memory footprint. Specifically, schemes that spawn a VM for a function invocation from a previously-generated snapshot (SEUSS [14], REAP [75]) do not enable the sharing of any of these categories. The same is mostly true for schemes that fork the execution of a function invocation from a template (SOCK [52] and Catalyzer's sfork [18]). Schemes that load the shared libraries into the container before forking a process to execute the function invocation (SAND [1] and process-based Nightcore [31]) enable the sharing of the *LibLd* footprint across function invocations. They save substantial memory.

In this paper, we note that there is still a substantial amount of memory footprint that can be shared across invocations of the same function: the read-only data that is function-specific and independent of individual invocations of the function (*Init*). The average footprint of such data in Figure 5 is 20.4 MB. Our proposal with MXContainer will be to delay the forking of a process for an invocation until such data is initialized once by a special process (Dispatcher). As a result, all function invocations will automatically share this data. Based on the numbers in Figure 5, this approach will allow us to keep in memory on average 3.4× more concurrent function invocations than process-based Nightcore. The result will be much higher concurrency and throughput.

3.2.4 There Are Concurrent Accesses to the Same Storage Locations. All the concurrent invocations of the same function execute the same code and, intuitively, should access the same storage area for the same or similar data at similar times. If this is true, there is an opportunity to merge the accesses to save I/O bandwidth. An analysis of production FaaS Azure traces [11, 64] shows that 12% of the applications access the same data blob in all of their invocations. Moreover, invocations access relatively few different blobs: 66% of the applications access less than 100 different blobs across all invocations. More importantly, a given blob is accessed in a bursty manner, offering opportunities for access merging. Figure 6 shows the CDF of the interarrival time of accesses to the same blob. It can be seen that 18% and 54% of the accesses to a given blob happen within 1ms and 10ms, respectively.

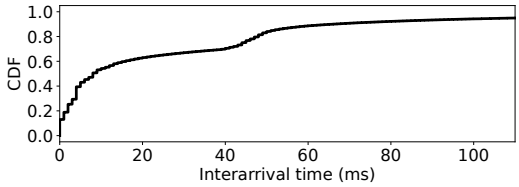


Figure 6: Inter-arrival time of accesses to the same blob.

In addition, blobs are typically small: 80% are smaller than 12KB. Hence, accessing many blobs in parallel, for the same or different data, creates a network bottleneck—not due to data volume, but due to connection overheads. Sharing and reusing connections for data transmission can reduce the bottleneck.

3.3 Implications

Our analysis has revealed a few key bottlenecks in serverless environments with bursty invocations of functions. First, functions are blocked on synchronous wait operations most of the time. Hence, unless cores are scheduled intelligently, the response time of function invocations can easily degrade substantially. Second, nodes issue similar requests to storage and invoke similar functions. The result is unnecessary I/O bandwidth consumption and pressure on storage and processors. Finally, containers consume substantial memory with replicated state. As a result, serverless systems are often limited by available memory.

4 MXFAAS OVERVIEW

To eliminate the bottlenecks uncovered in our characterization section, we now propose a new serverless platform design called *MXFaaS* or *Multiplexed FaaS*. MXFaaS optimizes execution during bursts of invocation requests for the same function—a typical occurrence in serverless environments. Unlike current platforms, MXFaaS leverages the synergies between these concurrent requests. More specifically, it efficiently multiplexes (i.e., shares) processor

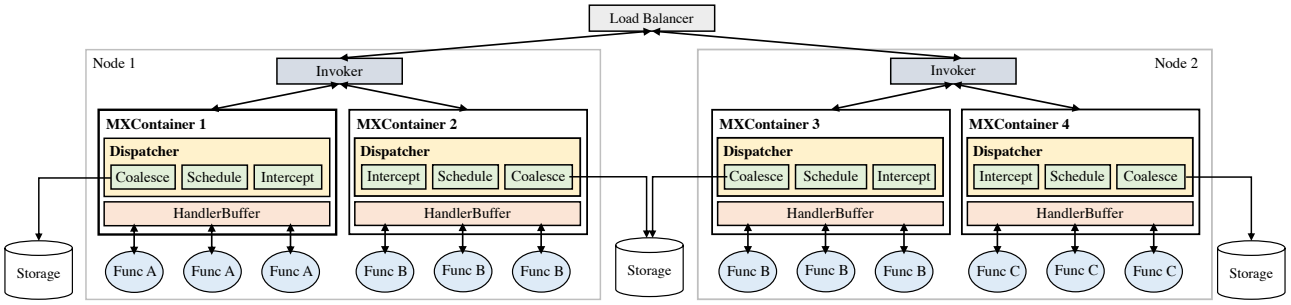


Figure 7: Overview of the MXFaaS serverless platform. The blue circles represent cores.

cycles, I/O bandwidth, and memory/processor state between concurrent invocations of the same function. The result is a higher throughput and lower latency serverless environment.

MXFaaS introduces a new container abstraction called *MXContainer* or *Multiplexed Container*, which can concurrently execute multiple invocations of the same function and owns a (potentially changing) set of cores. An MXContainer has a *Dispatcher* process and multiple *Handler* processes. The dispatcher initializes the container in the first function invocation. At every function invocation, the dispatcher forks a handler to serve the request. The multiple handlers concurrently execute invocations of the same function on different cores.

MXFaaS introduces three techniques, which improve the utilization of three key resources. First, to enable efficient use of processor cycles, the dispatcher in an MXContainer carefully suspends and resumes its handlers. Recall that a typical function execution is blocked most of the time, due to accesses to remote storage or to calls to other functions. Therefore, cores either remain idle for large periods or frequently context switch between containers of different functions. However, in an MXContainer, since the dispatcher has forked the handlers, the dispatcher can suspend and resume them. The dispatcher's aim is that, at any time, the OS can only schedule the *oldest N ready-to-execute invocations* of the function—where N is the number of cores currently assigned to the MXContainer. The dispatcher buffers the remaining set of invocations of the function, whose handlers may or may not be blocked on I/O or function calls. The result is efficient execution that minimizes average and tail response time.

Second, to enable efficient use of I/O bandwidth, the dispatcher combines remote storage accesses and remote function calls from multiple handlers running invocations of the same function. To support storage request combining, the dispatcher keeps a table with the outstanding storage accesses. When the dispatcher is about to issue a remote request, it checks the table and, if there is a matching request, it combines the two accesses. If there is no matching request, the dispatcher waits for some time before issuing the request—in the hope that an upcoming request can be combined with it. Combined storage requests can refer to a single key or to a vector of them.

In addition, the dispatcher combines function calls to the same function—for the same or different inputs. The overall result of combining storage accesses and remote function calls is lower network bandwidth needs and reduced pressure on storage and processors.

Third, to enable efficient use of memory/processor state, the dispatcher first initializes the MXContainer state and after that, on demand, spawns a handler process per function invocation. With this support, all invocations share the unmodified initialization state (*LibLd* plus *Init* in Section 3.2.3 and Figure 5)—while protecting their private data via copy-on-write. The result is a reduced memory footprint, which enables more containers to reside in memory at a time and, therefore, effectively reduces startup overhead.

Further, by executing these multiple invocations of the same function on the owned cores, the MXContainer also enables reuse of the cache and branch predictor state.

5 MXFAAS DESIGN

Figure 7 shows the MXFaaS serverless platform. It has a Load Balancer module and, in each node, an Invoker module and one or more MXContainers. An MXContainer manages the concurrent execution of multiple invocations of a function on a node, and owns a (dynamically changing) set of local cores. A node can have MXContainers for different functions, but at most only one for a given function. Different nodes may have MXContainers for the same function.

The dispatcher in an MXContainer admits requests for the function. It buffers those that are: (1) blocked on I/O and therefore unable to run, or (2) ready to execute but lack a core to run on. It only allows the OS to schedule as many ready-to-execute invocations of the function as cores the MXContainer owns. The dispatcher regularly informs the node's Invoker of its buffer's utilization.

When an Invoker observes that a local MXContainer becomes overloaded or underloaded, it dynamically changes the number of local cores assigned to it. When an MXContainer is overloaded and unable to get more cores, the global Load Balancer is informed. At that point, the Load Balancer allocates another MXContainer for the same function in another node. From then on, the Load Balancer dynamically shares the load between the two MXContainers.

In this section, we describe how MXFaaS supports the three techniques outlined in Section 4.

5.1 MXContainers for Sharing Processor Cycles

As shown in Section 3.2.2, a typical serverless function spends most of its time blocked, waiting for data from remote storage or for the return of an RPC. In some current systems, the OS does not preempt the blocked request because the FaaS platform has purposely limited the number of concurrently-running requests. In other systems, the FaaS platform allows over-subscription. Hence, the OS preempts the blocked request and schedules another request for the same or another function. Unfortunately, this operation is

inefficient without special support: the OS interleaves the execution of multiple containers without deliberately trying to complete older function requests first. The result is degraded average and tail response time.

The MXContainer approach solves this problem by having the dispatcher help manage the scheduling of the handlers. Recall that the dispatcher has spawned the handler processes and, hence, can suspend/resume them. In an MXContainer, the dispatcher maintains a buffer (*HandlerBuffer*) with the handlers that are ineligible to run. These handlers correspond to function invocations that: (1) are blocked on I/O or RPCs and therefore unable to run, or (2) are ready to run but are not the oldest N ready-to-execute invocations—where N is the number of cores currently owned by the MXContainer. Effectively, the dispatcher only allows the OS to schedule the handlers for the oldest N ready-to-execute invocations of the function; the rest are kept buffered in *HandlerBuffer*. This functionality minimizes average and tail response time.

This functionality is supported as follows. First, when the dispatcher initially forks a handler process for a request, the dispatcher (1) records the handler's sequence order and (2) if all the owned cores are busy, it suspends and buffers the handler in *HandlerBuffer*, marking it as *Ready*. Second, when a running handler reaches a blocking call, the call is redirected to the dispatcher, which buffers the handler in *HandlerBuffer*, marking it as *Blocked*. Finally, when the response for the remote storage access or RPC call is received, the dispatcher intercepts it, passes the value to the corresponding handler and, depending on the handler's sequence order, it either (1) keeps the handler suspended in *HandlerBuffer*, now marked as *Ready*, or (2) resumes this handler and suspends a younger, running handler of the same function. Again, the dispatcher can do this because it has spawned both handlers.

Figure 8 shows an example of this mechanism. Figure 8a shows a possible timeline of a function execution; the function spends some time waiting for I/O. Figure 8b shows the execution of six invocations of the same function in an MXContainer that owns two cores. The invocations are ordered based on arrival time from left to right. An invocation can be either using the CPU (*Busy*) or buffered in *HandlerBuffer* marked as *Blocked* or *Ready*. At time t_0, the dispatcher picks the two oldest invocations: *Invoc1* and *Invoc2*. At t_1, it picks *Invoc3* and *Invoc4* over *Invoc5* and *Invoc6*. At t_2 and t_3, it again picks older invocations over *Invoc5* and *Invoc6*.

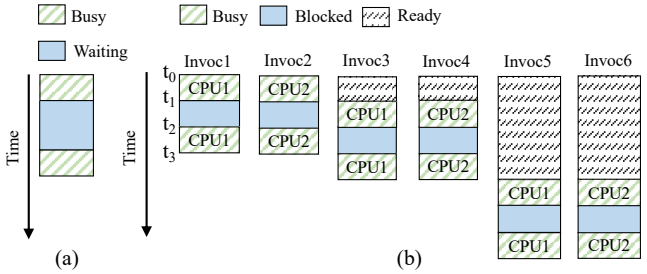


Figure 8: Interleaving of function invocations in two CPUs.

Using the Shortest Remaining Processing Time first (SRPT) algorithm can further reduce the response times when the execution time of the requests has a high variation. In practice, the requests of a given function in FaaS environments are of similar size and duration even when using different inputs [34, 67]. Moreover, SRPT requires estimating the remaining execution time. Consequently, we do not use SRPT.

Overall, in MXContainers, function invocations share processor cycles in a way that minimizes average and tail response time.

5.2 MXContainers for Sharing I/O Bandwidth

As multiple handlers in an MXContainer concurrently execute multiple invocations of the same function, these handlers are likely to issue requests for the same storage area (and potentially even the same keys). They are also likely to issue RPCs for the same functions, possibly even using the same argument values. Recall that the dispatcher intercepts all these blocking requests. This fact offers the ability to combine storage accesses or RPCs from multiple handlers—minimizing the network load and the pressure on storage and CPUs. We consider the two types of combining.

5.2.1 Remote Storage Access Combining. To combine remote storage accesses, the dispatcher keeps a software *Miss Status Holding Table* (MSHT). The MSHT has an entry for each outstanding storage access from this MXContainer. It is analogous to the hardware structure that records outstanding cache misses in cores.

When the dispatcher is about to issue a read to remote storage, it checks the MSHT. If there is already an outstanding read to the same key, the dispatcher issues no request. Instead, it combines the two read requests by augmenting the existing MSHT entry with additional information. When the key is received by the node, it will be passed to both the initial and the new requesting handlers.

If the dispatcher wants to issue a read and there is an outstanding write to the same key, or wants to issue a write and there is an outstanding access to the same key, the dispatcher delays its request until the previous access completes.

If the dispatcher is about to issue a remote storage access and does not find an existing entry in the MSHT for the key, it waits a certain time period (T_{merge}) before issuing the request. The goal is to coalesce the request with any subsequent requests to the same or different key that may come within a small time period—and therefore issue a single request instead of several. When the dispatcher combines requests for different keys, it issues one vectorized request to the remote storage. The MSHT records which handler accessed which key. When the dispatcher receives the response, it unfolds the vector and forwards the correct values to the appropriate reading handlers.

Figure 9 shows an example of accesses to different keys without coalescing (a) and with coalescing (b).

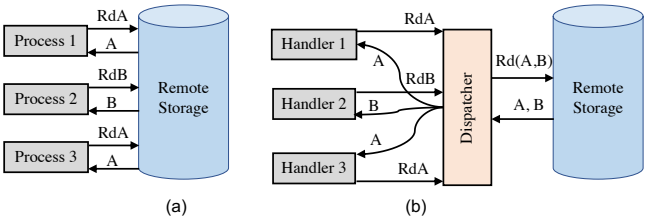


Figure 9: Storage access coalescing in MXFaaS.

When the load is low, delaying a request may not result in a merging event and, instead, can cause an increase in average and tail latency. Thus, the dispatcher monitors the load and dynamically decides whether to enable merging.

5.2.2 Function Call Combining. A similar strategy could be used to combine RPCs to functions. However, functions may have side effects, which means that the outcome of two calls with the same argument values may be different than the outcome of one single call. Hence, the safe approach to combining involves delaying the RPC for T_{merge} cycles and, if other RPCs to the same function are detected in the meantime (with or without the same argument values), bundle them all in a single I/O transaction that requires executing all the function calls at the destination node.

A special case is functions that, when invoked with the same inputs, produce the same outputs and have no side effects. If the programmer knows that a function behaves in this way, she can annotate the function as *pure*. For pure functions, the dispatcher maintains a table recording the set of {inputs, outputs} tuples observed in the past. When the dispatcher is about to call a pure function with certain input values, it checks the table. If it finds an entry with the same inputs, it reads the outputs and skips the RPC. Pure functions are common: in the SeBS [15], TrainTicket [83], and FunctionBench [35] benchmark suites, 50.0%, 57.6%, and 60.0% of the functions, respectively, are pure.

5.3 MXContainers for Sharing Memory and Processor State

The MXContainer for a function instance has a dispatcher process and multiple handler processes. In the first invocation of the function, the dispatcher first executes the function initialization. In every invocation of the function, including the first one, the dispatcher forks a handler process that executes the function. With this design, the different handlers automatically share the unmodified initialization state (*LibLd* and *Init* in Section 3.2.3) and, on a write, create private page copies via copy-on-write. This is in contrast to previous FaaS schemes, where different invocations of the same function share at most *LibLd*. As shown in Figure 5, *Init* is large. When many handlers are running concurrently, sharing *Init* pages rather than replicating them reduces the memory footprint significantly. As a result, the MXContainer approach substantially reduces the total memory footprint of the multiple concurrent invocations relative to previous FaaS schemes. The smaller footprint frees-up space for containers of other functions.

With MXContainers, the startup overhead of the multiple concurrent invocations is reduced, as it is paid only once for the first invocation of the burst. Given the short-lived execution of functions, reducing the startup overhead speeds-up execution significantly. Note that, in an MXContainer, each function invocation is executed in the address space of a new process. No process is reused to execute multiple function invocations. Thus, MXContainers avoid the security and correctness issues of reusing a process for multiple invocations (Section 2).

Serverless functions do not typically write to files because their updates are not persistent. However, they could read from read-only files or write to temporary files. Hence, if we allow multiple processes to run concurrently inside the container, we need to ensure there are no data races in file updates. To achieve this, we develop a scheme similar to copy-on-write memory pages. As long as a handler process only reads from a file, it can use the shared initial file. However, once the handler tries to perform an update to the file, it creates its own temporary file, with a unique name. From this moment on, all reads and writes by the handler are done on the new temporary file. When the handler completes its execution, all of its temporary files are discarded.

An alternative would be to use existing container primitives such as `namespace` and `chroot`. However, these primitives are inefficient because they require copying all the files before the handler starts execution.

Finally, since all these multiple invocations of the same function run on the cores owned by the MXContainer, their processes reuse the cache and branch predictor state with each other. Individual functions typically have low divergence in the set and order of executed instructions across different invocations (even with different inputs) [34, 67]. Thus, the MXContainer design significantly reduces the misses in caches and branch predictors.

6 MXFAAS IMPLEMENTATION

We build MXFaaS in both OpenWhisk [5] and KNative [37], two serverless cloud platforms. In this section, we discuss a few important implementation aspects.

6.1 Function Runtime

We implement the MXFaaS runtime with 1.2K lines of Python code. Users can write functions in any language that supports the forking mechanism. The initialization of a function is performed by importing a module (for Python functions) or by loading a shared library (for C/C++/Rust functions). We discuss support for additional languages in Section9.

As indicated in Section5.1, the dispatcher in an MXContainer intercepts the blocking calls in handlers. A function can employ various library APIs to invoke other functions or to perform I/O. In Python, most communication libraries such as `requests`, `redis`, `minio`, `pymongo`, and `boto3` call APIs from the `recv` family of Python's `socket` module (e.g., `recv`, `recv_from`, `recv_into`) to block. Hence, we overload all these `socket` APIs with wrappers that inform the dispatcher when a handler (identified by its PID) calls a block API. When the dispatcher is notified, it calls the socket API on behalf of the handler (after suspending the handler). Later, the dispatcher receives the response and informs the handler.

To be able to support other languages, the dispatcher needs to intercept blocking calls beyond Python's `socket` module.

We inspect blocking I/O and RPC libraries from different languages and find that they eventually invoke the `recvfrom` system call. To capture `recvfrom`, MXFaaS uses `LD_PRELOAD` [58] to intercept target system calls in user mode. When a `recvfrom` system call is captured, the wrapper forwards the handler PID and call arguments to the dispatcher. The rest of the algorithm remains unmodified. In over 110 open-source functions analyzed, we did not observe any other blocking calls that are long enough to be exploited. There are some local OS blocking calls, but these operations are too short to be exploited for scheduling within an MXContainer.

We implement copy-on-write for files by intercepting system calls for filesystem operations, such as `open`, `read`, and `write`. As long as the handler does not perform updates to the file, it can read from the original shared file. Once it tries to update the file via the `write` call, we copy the initial file to a temporary file and save the

translation from the initial file name to the newly-created file name. All later operations to the file are redirected to the temporary file.

We implement the I/O access combining support for Redis. Specifically, the dispatcher intercepts `get` and `put` requests by handlers. For a `get`, if there is an outstanding `get` for the same key, the dispatcher augments the existing MSHT entry with new information. Then, when the dispatcher receives the response, it forwards it to all handlers that requested it. Otherwise, the dispatcher coalesces multiple requests to different keys issued within T_{merge} into one collective request. It sends one `mget`/`mput` request instead of many `get`/`put`. Other storage services can be supported in similar ways.

6.2 Serverless Platform

MXFaaS requires platform modifications to set the number of MXContainers in the system and the number of cores for each MXContainer. Initially, the load balancer picks a node for each MXContainer. In a given node, MXFaaS divides the cores among different MXContainers based on their relative needs. To estimate the core needs of MXContainers, MXFaaS dynamically measures: (1) the fraction of requests for each type of function and (2) the time that handlers spent buffered in state *Ready* in the HandlerBuffer of each MXContainer. Based on these measurements, MXFaaS sets (and dynamically adjusts) the number of MXContainers in the whole platform for each function and the number of cores assigned to each MXContainer.

Consider the MXContainer of a function in a node. Let C be the number of cores in the node, R the overall number of function requests per second (RPS) received by the node, and F the RPS for the function supported by the MXContainer. Then, the MXContainer is assigned $max(C \times \frac{F}{R}, 1)$ cores in the node. At the same time, MXFaaS monitors the average amount of Ready time per function invocation in each MXContainer. It checks such time against two thresholds: a low one (*LowReady*) and a high one (*HighReady*). If the average Ready time in an MXContainer is higher than *HighReady*, MXFaaS first tries to get more cores for the MXContainer by stealing local cores from another MXContainer whose average Ready time is less than *LowReady*. If the MXContainer is unable to get the necessary local cores, MXFaaS creates a new MXContainer for the same function in another node.

To deal with transient loads, MXFaaS sets aside a pool of idle cores on every node (i.e., server). When an MXContainer experiences a load spike, the invoker first takes cores from the pool before stealing cores from other containers in the node. When the load for the container drops, it returns cores back to the pool. The dispatcher observes if the load changes quickly and, if so, it can further reduce *LowReady* to prevent a container from returning cores too soon.

We changed the implementations of OpenWhisk's invoker and load balancer [54, 55]. The invoker works with MXContainers in addition to with traditional containers: it is informed of the MXContainer load and allocates CPU cores accordingly. The load balancer is informed of any MXContainer overload. The modifications required about 400 lines of Scala code.

For KNative, we modified the autoscaler and activator [38, 39], which play a similar role as the load balancer and invoker in OpenWhisk. The modification is identical to that of OpenWhisk but written in about 300 lines of Go code.

6.3 Multitenancy and Security Implications

MXFaaS follows the multitenancy security model of existing serverless platforms [10]: a container belongs to a tenant, and different end-users can issue service requests that can be executed in the same container without special security protections.

Requests executed in different MXContainers do not share any state and run on different cores. In this paper, we assume that it is safe to collocate multiple MXContainers from different tenants in the same server. Requests executed in the same MXContainer use process-level isolation: they share initialization state but cannot access each others' private data. Moreover, they execute on the same cores. Therefore, it is potentially easier for them to use shared hardware resources such as MSHRs, branch predictors, and caches as side channels. Most of these side channels already exist in current systems. An analysis of the resulting security implications is beyond this work's scope.

7 METHODOLOGY

Evaluation Environment. We evaluate MXFaaS on OpenWhisk and KNative in a 15-server cluster. Each server has an AMD EPYC 1-socket 7402P processor with 24 cores (2-way multi-threaded), 128GB DRAM and a 128MB LLC. Each server runs Ubuntu 20.04.2 LTS. In this paper, we only discuss the results from OpenWhisk. The KNative results are similar because MXFaaS is not specific to the underlying system.

Baseline System. To serve as baseline, we have emulated the state-of-the-art Nightcore [31] on top of OpenWhisk and KNative. Each container can support up to a maximum number of process-based invocations of the same function. The processes are forked when the libraries are loaded—i.e., before the function initialization. Therefore, unlike MXFaaS, processes can only share the *LibLd* state in Figure 5. If there are more concurrent invocations than the maximum allowed, the additional requests are buffered and run later when some of the previous invocations complete.

Evaluated Functions and Applications. We use functions from FunctionBench [35], a suite that includes ML training, ML model serving, and image/video processing. We choose FunctionBench because it is widely used in prior serverless research [4, 24, 44, 75, 81, 82]. Since FunctionBench does not include functions from the popular web services, we include two standalone web functions from TrainTicket [68], a large serverless application suite (CreateOrd and PayOrd). Web-service functions are more lightweight than those in FunctionBench (Figure 4a). The complete set of functions evaluated is in the upper part of Table 1. We also use serverless applications composed of several functions that call each other. We select four representative applications from TrainTicket [68] (lower part of Table 1). We use Redis [62] as the storage service for all the evaluated functions. To be conservative, we annotate no function as pure (Section 5.2.2).

Workloads. We evaluate MXFaaS under *low*, *medium*, and *high* load levels, corresponding to an average of 450 requests per second (RPS), 1000 RPS, and 1800 RPS, respectively. We choose these low, medium, and high load levels based on the fact that they drive the CPU utilization in our MXFaaS environment to ≈ 25%, 50%, and 70%, respectively, which is representative [16, 27, 47, 63]. Also, like

Table 1: Serverless benchmarks used in the evaluation.

Benchmark	Description
Standalone Functions	
LR-serv	ML model serving: Logistic regression
CNN-serv	ML model serving: CNN-based image classification
RNN-serv	ML model serving: RNN-based word generation
ML-tr	ML model training: Logistic regression
VidConv	Video processing: Apply gray-scale effect
ImgRot	Image processing: Rotate image
ImgRes	Image processing: Resize image
CreateOrd	Web service: Write created order to database
PayOrd	Web service: Withdraw money from account
Serverless Applications	
TcktApp	Get all tickets for a given trip (15 functions)
TripInfApp	Get information about the trip (24 functions)
GetLeftApp	Get unsold tickets for a given time frame (5 functions)
CancelApp	Cancel an order (4 functions)

in prior research on serverless systems [1, 12, 26, 69, 72, 74, 81], we use the Poisson distribution to model request inter-arrival time.

Parameter Setting. We perform sensitivity analyses to determine the values of MXFaaS parameters. For T_{merge}, Section 8.2.3 selects 1ms. Moreover, we set the SLO of a request to 2× the execution time of the same request on an unloaded system. This is more conservative than the prior art [12, 26]. When the average response time gets close to 1.5× the unloaded execution time, MXFaaS considers the corresponding MXContainer to be getting overloaded. Thus, we set *HighReady* to 40% of the function execution time. When the average response time is close to the unloaded execution time, MXFaaS considers the corresponding MXContainer to be underloaded. Thus, we set *LowReady* to 10% of the function execution time.

An MXContainer only accepts a certain number of concurrent function invocations. Such number depends on the number of cores it owns (N), the average busy (B) and idle (I) time of an invocation, and the *HighReady* threshold. Specifically, the execution time of a request is I+B. Within the I period, we can squeeze in $\frac{I}{B}$ additional requests. Therefore, the total number of requests executing in N cores is $N \times (1 + \frac{I}{B})$. If we are willing to add *HighReady* delay to each I+B execution without violating the SLO, the response time becomes I+B+*HighReady*. If the number of requests to get the response time I+B is $N \times (1 + \frac{I}{B})$, then using a simple proportion we can derive that the number of requests to satisfy the response time I+B+*HighReady* is $N \times (1 + \frac{I}{B}) \times (1 + \frac{HighReady}{I+B})$. Of these, N are running and the rest are queued in the HandlerBuffer. An MXContainer dynamically targets this number of queued requests. The dispatcher informs the invoker about the number of queued requests every 200ms. If the queue goes over this number, the dispatcher immediately tells the invoker to either provide extra cores or offload some of the future requests to another server.

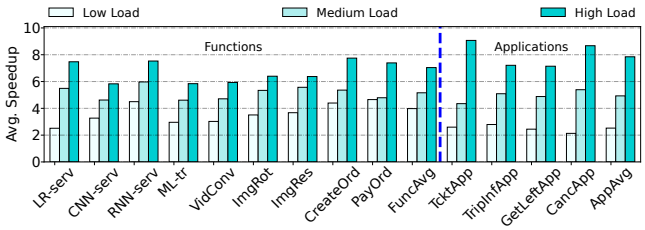
8 EVALUATION

In this section, we evaluate MXFaaS' end-to-end latency reduction, its resource efficiency, its scalability, and a comparison to proactive container creation techniques.

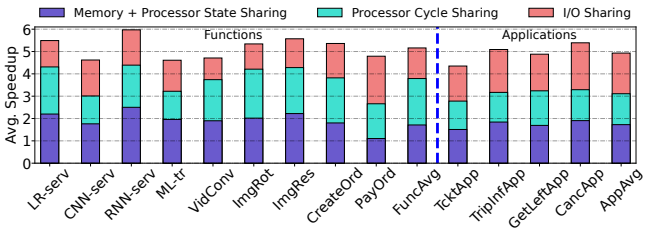
8.1 End-to-end Latency Reduction

We measure MXFaaS' ability to reduce the end-to-end latency of requests that invoke serverless functions or applications. The end-to-end latency of a function or application invocation is the time from when the client sends a request until when it receives the result. We normalize the MXFaaS latency to the latency with Nightcore [31], which is our state-of-the-art baseline.

8.1.1 Average Speedups. Figure 10a shows the inverse of the normalized latency, namely, the speedups of MXFaaS over the baseline, for low, medium, and high system loads. From left to right, the figure shows bars for the functions, their average, the applications, and their average. On average across all benchmarks and load levels, MXFaaS delivers a speed-up of 5.2×.



(a) Speedups for various loads.



(b) Speedup breakdown aggregated across all three load levels.

Figure 10: Speedups of MXFaaS over Nightcore.

MXFaaS achieves higher speedups with higher loads because more requests benefit from the multiplexing. Under high load, the latency of baseline increases substantially, as the baseline does not exploit the idle times per request and, therefore, supports limited parallelism. The result is an inefficient use of processor cycles and queuing of ready requests even though CPUs are idle. This queuing effect is amplified in short-lived functions. For example, the web-service functions (CreateOrd and PayOrd) have a high baseline overhead (as they have the shortest execution) and thus they benefit substantially from MXFaaS.

Figure 10b shows the contributions of each of the three MXFaaS components to the average speedup. The numbers are aggregated across all three load levels. We apply the three components one by one: sharing memory and processor state (Section 5.3), sharing processor cycles (Section 5.1), and sharing I/O (Section 5.2). All the techniques are effective. They deliver average speedups of 1.9×, 1.9×, and 1.4×, respectively. Processor cycle sharing especially helps functions with relatively longer idle time due to blocking. I/O sharing has higher contributions in serverless applications, where functions have smaller data volumes and more communication. Finally, memory and processor state sharing especially helps ML functions that share a large model and a large input dataset.

8.1.2 Tail Latency. MXFaaS significantly reduces the tail latency of the function/application requests. Figure 11 shows the P99 tail latency in MXFaaS for different loads normalized to that in the baseline. On average across all benchmarks and loads, MXFaaS

reduces the P99 tail latency by 7.4×. As the load increases, the reduction also increases. In the baseline, the tail latency is high due to queuing effects when requests are waiting for resources. MXFaaS reduces the tail latency in two ways. First, it only lets the OS schedule *the oldest N ready-to-execute invocations* of a function, where *N* is the number of cores owned by the function's MXContainer (Section 5.1). Second, it is able to use memory and processor state more efficiently.

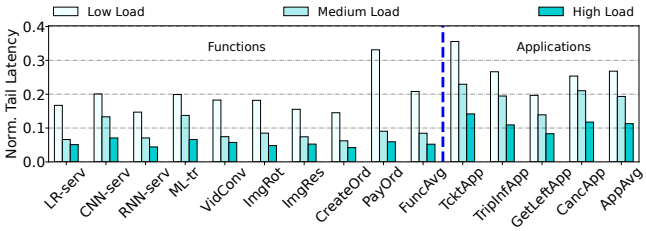


Figure 11: Normalized P99 tail latency.

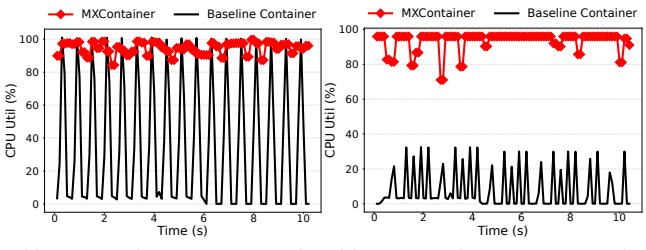
8.2 Resource Efficiency

MXFaaS significantly improves resource efficiency, which results in higher throughput. In this section we consider several aspects.

8.2.1 Container CPU Utilization. We compare the CPU utilization in MXContainers and in the baseline containers. Figure 12 shows the container CPU utilization over time in MXContainers and in the baseline, while executing CNN-serv (the least idle workload) and CreateOrd (the most idle workload). We see that the CPU utilization of the MXContainer is around 90-100% most of the time, thanks to efficiently multiplexing many function invocations in the container. The CPU utilization of the baseline container is highly fluctuating and often very low.

Since the MXContainer already drives the system to near 100% container CPU utilization, accepting more ready function invocations to compete for cores would only lead to CPU contention. Such contention would degrade performance. To validate this point, we conducted a sensitivity analysis by allowing the OS to schedule more ready-to-run function invocations than the amount of cores owned by the MXContainer (Section 5.1). It can be shown that, allowing 20% and 50% more ready requests to contend for scheduling, increases the tail latency by 1.6X and 4X, respectively.

8.2.2 System Throughput. The higher CPU and memory efficiency results in improved system throughput. We define System Throughput as the number of concurrent requests that the system can process before the average response time becomes twice that of an unloaded system. We show the value in Table 2, based on workload classes. MXFaaS increases the average throughput by 4.8× over the baseline.



(a) CNN-serv (least idle workload) **(b) CreateOrd (most idle workload)**

Figure 12: Container CPU utilization over time.

Table 2: System throughput in MXFaaS and in the baseline.

Workloads	Baseline (Req/s)	MXFaaS (Req/s)	Improvement (Times)
ML-functions	900	4100	4.6
Img/Video Processing	1250	6000	4.8
WebServices	1800	9000	5.0
TrainTicket Apps	200	900	4.5
Average	1037.5	5000.0	4.8

8.2.3 I/O Bandwidth Savings. We measure the effect of MXFaaS' I/O sharing technique. Figure 13 shows, for Baseline and MXFaaS, a histogram of the latency to fetch the data from global storage. The figure corresponds to the ImgRot function under the high system load. The two designs that we compare include the recently-proposed caching scheme in [64]. From the figure, it can be shown that MXFaaS reduces the median latency from 0.49s to 0.34s. The effect on the tail latency is even more substantial: the latency decreases from 5.81s to 1.76s. The reason is that I/O combining relaxes the pressure on the network and on remote storage. We also see that a large number of MXFaaS requests have very low latency, as a result of hitting in the MSHT (Section 5.2).

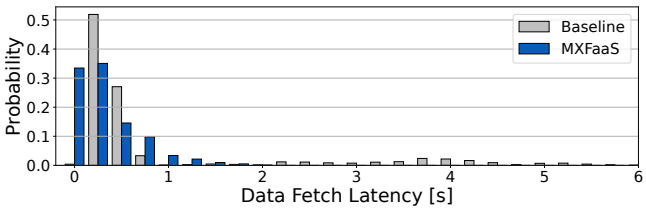


Figure 13: Histogram of data fetch latency for the ImgRot function.

To pick the value of T_{merge}, we performed a sensitivity study. As we increase T_{merge}, the fraction of merged I/Os also increases. However, the tail latency of the data fetches also increases. We pick a T_{merge} value equal to 1ms, which merges substantial requests without affecting the tail latency much. Figure 14 shows the fraction of merged I/Os and the tail latency of data accesses for the ImgRot function and the high load, as we vary T_{merge}. We see that, for the chosen T_{merge} value, MXFaaS merges 46% of I/Os.

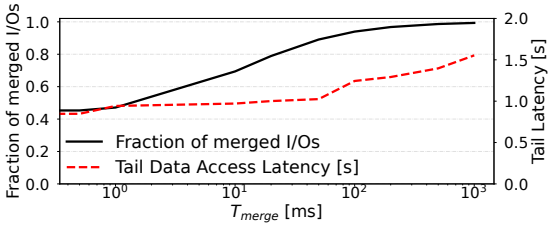


Figure 14: Sensitivity study of T_{merge} for the ImgRot function.

The percentage of merged I/Os depends on the data fetch latency and the system load. Across all applications and loads, MXFaaS reduces the number of I/Os by 24%-83%, with an average of 52%. On

average, a pending request in the MSHT combines with 6.1 other requests. Further, a request stalling for T_{merge} combines with 3.2 subsequent requests.

8.2.4 Memory/Processor State Reuse. The MXContainer design enables substantial sharing of memory and processor state across invocations of the same function. Figure 15 shows the average memory footprint in baseline and in MXFaaS across all the three loads. In the figure, the bars are normalized to the footprint in MXFaaS. The numbers on top of the bars are the absolute values of the footprint in Baseline and MXFaaS in GBytes. From the figure, it can be shown that MXFaaS reduces the average memory footprint of the functions and applications by 3.4× (from 67.2GB to 19.5GB). Higher loads lead to higher reductions.

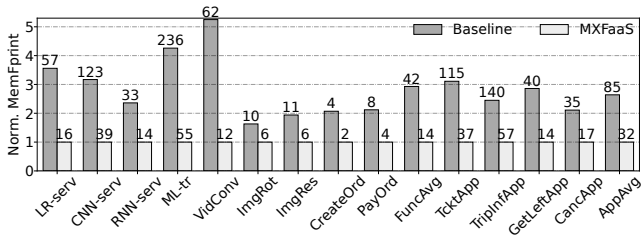


Figure 15: Normalized memory footprint in baseline and in MXFaaS averaged across all three loads. The numbers on top of the bars are the absolute values in GB.

To reason about the branch and cache state reuse, we use the hardware performance counters [59] of the servers to measure the number of misses in the branch predictor and in the caches. We consider two cases: MXFaaS deliberately schedules the requests for the same function on the same set of cores (Case 1), and MXFaaS lets the OS schedule the requests on any currently available core (Case 2). Figure 16 shows the measured Misses Per KInstruction (MPKI) in L1 caches, L2 caches, and branch predictor, and the average response time of requests in Case 1 normalized to those in Case 2. The figure shows data for each function. On average, MXFaaS reduces the L1, L2 and branch MPKI by 46%, 43%, and 45%, respectively, which translates into a 30% reduction in the response time.

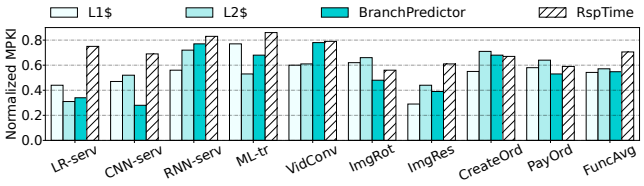


Figure 16: Microarchitectural state reuse in MXFaaS and its impact on the response time of requests.

8.3 Scalability

MXFaaS is a scalable serverless platform. We conduct a scalability experiment with three cluster sizes: 10, 15 (default) and 20 servers. Figure 17 shows the speedup of MXFaaS over baseline across all benchmarks with medium load for the three cluster sizes. As the cluster size increases, MXFaaS achieves higher relative speedups over the same-size baseline. On average, MXFaaS speeds up the execution by 4.4×, 5.2× and 6.1× with 10, 15 and 20-server clusters.

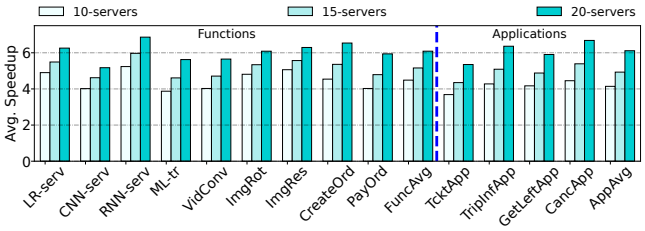


Figure 17: Speedup of MXFaaS over the baseline for different cluster sizes.

8.4 Comparing to Proactive Container Creation

There are several techniques that reduce FaaS startup-time by predicting which containers will be needed next, and proactively allocating and preparing them [12, 17, 26, 66, 70]. Instead of comparing MXFaaS with each technique individually, we compare MXFaaS with the *best-case scenario*: the prediction technique is 100% correct and there is no cold-start time—if there is enough memory space for the container.

Figure 18 compares MXFaaS with this best-case scenario. It shows the average response time of functions under low, medium and high loads. We show a representative function (CNN-serv) and application (TcktApp). On average across the three loads, MXFaaS reduces the response time over this ideal scheme by 1.6× for CNN-serv and 1.7× for TcktApp. Across all benchmarks and loads, the reduction is 2.1× (or 2.9× if we only consider high load). There are two reasons for the baseline's losses. First, under medium and high loads, available memory becomes scarce. Hence, some requests need to wait to allocate a container till some memory is freed-up. MXFaaS is not as constrained by memory (Section 8.2.4). Second, under any load, MXFaaS benefits from processor cycle and I/O bandwidth sharing.

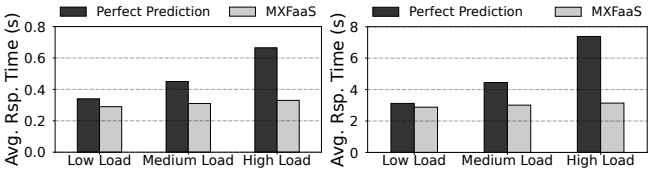


Figure 18: Average response time of two benchmarks in MXFaaS and in an ideal environment with perfect pre-warming.

9 DISCUSSION

It is straightforward to support the execution of function invocations as threads in an MXContainer instead of processes. We implement such threading support and measure its performance. Replacing forked processes with threads improves the performance of our benchmarks by 11% on average. However, threads (i) have weaker isolation and (ii) require the function implementations to be thread safe. As indicated by AWS [8], the thread model raises correctness issues in functions that modify global variables.

Languages such as Java and NodeJS do not support the fork semantics. For these languages, the MXContainer dispatcher cannot fork handlers but needs to prepare the initialization of each handler,

for example, by loading modules. We are working on additional support for these languages. One can choose to use thread-based MXContainers for these languages if not providing strong isolation and thread-safety are not concerns.

10 RELATED WORK

10.1 Serverless Systems

Work on serverless systems falls into four categories.

Snapshotting. SEUSS [14] and REAP [75] reduce the cold-start overhead by spawning a VM from a previously-generated snapshot. These techniques attain large overhead reductions, although they still have startup times of 70-1000ms. They are not optimized for high concurrency. As the number of concurrent instances increases, the startup overhead becomes larger due to the snapshot-reading contention. They do not exploit the requests' idle time.

Fork from a template. SOCK [52] and Catalyzer's sfork [18] rely on the assumption that containers for different functions have a lot of data in common. Hence, they create a new container to serve a request by forking from a template container shared across all functions. Then, they insert function-specific code in the forked container, execute the function-specific initialization and only then execute the handler. These schemes reduce, but not remove, the cold-start overhead. They hurt performance by executing function-specific initialization code for every concurrent invocation. In addition, they do not share read-only function-specific initialization data, which often has a large footprint. They do not optimize scheduling or manage concurrency. Finally, SOCK requires a special container type, while sfork in Catalyzer requires OS modifications.

Thread/process-level isolation. These schemes use different abstractions for function execution, including container [5, 29, 37], process [1, 40, 52], thread [31, 37], and software-based fault isolation [13, 71]. To reduce cold-start overhead, some schemes relax the isolation boundaries and, in a given container, allow the execution of multiple invocations of the same function (Nightcore [31]) or the execution of the different functions of an application (SAND [1] and Faastlane [40]).

The approach used by SAND and Faastlane does not handle efficiently the common case of a function that is shared among multiple applications. First, the shared function cannot scale independently of other functions in the application. Second, the shared function needs to be copied to all the containers that serve the different applications. Another shortcoming of including all the functions of an application in a container is that the size of the container is very large, as it has to include the support for potentially different runtimes, languages, libraries, and packages. Furthermore, functions can come from different security domains, and executing them together in the same container may leak.

Nightcore executes concurrent invocations of the same function in a container with separate processes or threads. It provides sub-optimal concurrency management because (1) a container allows only a predetermined number of requests to be dispatched and (2) the container queues requests above this threshold until prior requests are completed. Nightcore, like the previous two schemes, does not exploit the fact that a function spends significant time idle and thus, wastes available CPU resources. Finally, Nightcore forks processes as soon as the libraries are loaded—hence, processes are unable to share all the read-only function initialization data that is independent of individual invocations of the function.

Predictions. Some startup-time reduction techniques [12, 17, 26, 66, 70] predict which containers will be needed next, and proactively allocate and prepare them. In practice, accurate prediction is hard. Unless the accuracy is high, the response time grows significantly and resources are wasted. Further, when the load is high, available memory becomes scarce. Hence, even with high prediction accuracy, some requests need to wait to obtain memory.

10.2 Other Related Work

Microsecond-scale core allocation and scheduling. Recent works have developed μs-scale core allocation and scheduling techniques to improve CPU efficiency [23, 33, 57, 61]. They aggressively reallocate cores to minimize idle time. Unlike MXFaaS, they are agnostic to serverless workloads and do not consider the synergies between concurrent function invocations as MXContainer does. Many of them require OS changes, while MXFaaS does not.

Optimization of I/O and RPC. Many optimizations have been developed to reduce the overheads of storage I/O and RPC invocations of functions. They include data caching to reduce remote storage accesses [36, 45, 60, 64, 73, 77] and minimizing RPC overhead [31, 49]. MXFaaS' I/O access coalescing is complementary to these efforts, as it reduces I/O bandwidth and communication overhead originating from the containers. Other work [28, 43, 76] reduces the cost of distributing container images under bursty workloads. MXFaaS reduces data volumes as all the invocations of the same function in a given server share the container image.

Startup-time reduction. Some proposals reduce startup latency by proactively preparing function containers to hide latency [12, 17, 26, 66, 70], keeping containers warm [46, 70], or using snapshots and caches [14, 18, 52, 75]. While MXFaaS is complementary to these techniques, it does not benefit as much from them as other systems, since these techniques mostly impact only the first function invocation of the MXContainer.

11 CONCLUDING REMARKS

In this paper, we introduced MXFaaS, a new serverless platform design where concurrently-executing invocations of the same function share processor cycles, I/O bandwidth, and memory/processor state. MXFaaS introduces the new *MXContainer* abstraction, which enables substantial improvements in processor, I/O, and memory efficiency in serverless environments. Our evaluation showed that, with MXFaaS, serverless environments are much more efficient. Compared to a state-of-the-art baseline, MXFaaS on average sped-up the execution by 5.2×, reduced the P99 tail latency by 7.4×, and improved the throughput by 4.8×. In addition, it reduced the average memory usage by 3.4×.

ACKNOWLEDGMENTS

This work was supported in part by NSF under grants CNS 1956007 and CCF 2107470; by ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA; and by the IBM-Illinois Discovery Accelerator Institute. We thank Feiran (Alex) Qin for helping with the MXFaaS implementation on KNative.

A ARTIFACT APPENDIX

A.1 Abstract

Our artifact includes the prototype implementation of MXContainers, integration of MXFaaS with KNative, the scripts to perform the characterization study on the open-source production-level traces and serverless benchmarks, and the experiment workflow to run these workloads.

A.2 Artifact Check-list (Meta-information)

- **Program: MXContainers, MXFaaS atop of KNative**
- **Run-time environment: Ubuntu 18.04 or higher, Docker, Kubernetes (minikube for testing), KNative**
- **Hardware: Intel or AMD processors**
- **Metrics: Latency, throughput, CPU and mem. utilization.**
- **Experiments: Characterization, KNative implementation**
- **How much disk space required: 32GB**
- **How much time is needed to prepare workflow: 2h**
- **How much time is needed to complete experiments: 1h**
- **Publicly available: Yes**
- **Code licenses: MIT License**

A.3 Description

We have two main software artifacts.

First, we provide scripts to reproduce our characterization study. The scripts include the analyses of (i) the open-source production-level traces from Azure, and (ii) the open-source serverless benchmarks from FunctionBench. The scripts analyze (i) the request burstiness in serverless environments, (ii) the idle time of serverless functions, (iii) the breakdown of memory footprint of serverless functions, and (iv) the bursty access pattern in to the remote storage.

Second, we provide our implementation of MXFaaS: a novel serverless platform built upon KNative. MXFaaS includes two main components: (i) MXContainers that support efficient CPU, I/O and memory sharing across invocations of the same function, and (ii) Node Controller that supports core assignment across collocated MXContainers and extends auto-scaling features.

A.3.1 How to Access. The source code and benchmarks are hosted on Github: `jovans2/MXFaaS_Artifact`.

A.3.2 Hardware Dependencies. This artifact was tested on Intel (Haswell, Broadwell, Skylake), and AMD EPYC processors: Rome, Milan. Each processor has at least 8 cores.

A.3.3 Software Dependencies. This artifact requires Ubuntu 18.04+, Docker 23.0.1, minikube v1.29.0, and KNative.

A.4 Installation

First, clone our artifact repository:

```
git clone \
https://github.com/jovans2/MXFaaS_Artifact.git
```

Setting up the environment. In the main directory of the repository, script `setup.sh` installs all the software dependencies.
Execute: `./setup.sh`.
The script will first install Docker and set up all the required privileges. Then, it will install minikube, as a local Kubernetes, convenient for testing purposes. Finally, it will install KNative. The script will ask twice to choose one of multiple options. In both times choose the default value.
Once the installation is completed, open a new terminal and execute the following command `minikube tunnel`.

Downloading open-source production-level traces. To reproduce our characterization study we need open source traces from the Azure's production workload. We need
(i) `Azure Functions Blob Access Trace`, and
(ii) `Azure Functions Invocation Trace`.

Download the traces in the `characterization` directory of our repository by running `./download-traces.sh`.

Installing application specific libraries. To locally install all the libraries needed by our Python applications,
Execute: `./install-libs.sh` in the characterization directory.

A.5 Experiment Workflow

Characterization study. After the Azure's traces are in the characterization directory, to run all of our characterization experiments you need to execute `./characterize.sh`. This script will first analyze the request burstiness and bursty storage access pattern from Azure's traces (Figures 2 and 5), then it will analyze the serverless benchmarks from `functions` directory (Figures 3 and 4).

KNative prototype. Running `./deploy.sh` in the KNative prototype directory deploys the target functions as MXContainers on KNative. To test if all functions are successfully deployed, run `kn service list`. It should show all functions and their URLs. After 2 minutes, the flag `READY` should be set to `True`. Each function can be invoked with `curl <ip-addr>`. To test all functions at once, run `python3 knative-all.py`.
Next, we need to test the performance of MXContainers. There are three loads we test: Low, Medium and High. All loads use the Poission distribution. The scripts to run the experiments are located in the `experiments` directory. To run all the experiments at once execute `python3 run-all.py`.

A.6 Evaluation and Expected Results

Characterization study. Running `./characterize.sh` in the characterization directory, creates the following output files: `azure_burstiness.png`, `azure_blobs.png`, and `funcs.txt`.

The first two figures are Figures 2 and 5 from our paper. The third file contains the idle time and memory breakdown for each of the evaluated functions. The expected output is in `expected-output-funcs.txt`.

Latency measurements. Running `python3 run-all.py` in the `experiments` directory produces the output file: `run-all-out.txt`. The file contains for each of the tested functions and for each of the tested loads: average latency, median latency and tail latency. The reference output is in the `expected-output-all.txt`.

A.7 Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-badging
- http://cTuning.org/ae/submission-20201122.html
- http://cTuning.org/ae/reviewing-20201122.html

REFERENCES

[1] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. 2018. SAND: Towards High-Performance Serverless Computing. In *2018 USENIX Annual Technical Conference (USENIX ATC '18)*.

[2] Amazon AWS. 2023. AWS Lambda. https://aws.amazon.com/lambda/.

[3] Lixiang Ao, Liz Izhikevich, Geoffrey M. Voelker, and George Porter. 2018. Sprocket: A Serverless Video Processing Framework. In *Proceedings of the 2018 ACM Symposium on Cloud Computing (SoCC '18)*.

[4] Lixiang Ao, George Porter, and Geoffrey M. Voelker. 2022. FaaSnap: FaaS Made Fast Using Snapshot-Based VMs. In *Proceedings of the Seventeenth European Conference on Computer Systems (EuroSys '22)*.

[5] Apache OpenWhisk. 2023. https://openwhisk.apache.org/.

[6] AWS. 2023. AWS Lambda Anti-Patterns: Lambda functions calling Lambda functions. https://docs.aws.amazon.com/lambda/latest/operatorguide/functions-calling-functions.html.

[7] AWS. 2023. AWS Lambda Anti-Patterns: Synchronous waiting within a single Lambda function. https://docs.aws.amazon.com/lambda/latest/operatorguide/synchronous-waiting.html.

[8] AWS. 2023. AWS Lambda: Comparing the Effect of Global Scope. https://docs.aws.amazon.com/lambda/latest/operatorguide/global-scope.html.

[9] AWS. 2023. AWS Samples: AWS Serverless Workshops. https://github.com/aws-samples/aws-serverless-workshops/.

[10] AWS. 2023. Security Overview AWS Lambda. https://docs.aws.amazon.com/pdfs/whitepapers/latest/security-overview-aws-lambda/security-overview-aws-lambda.pdf.

[11] Azure. 2023. Azure Public Dataset. https://github.com/Azure/AzurePublicDataset.

[12] Vivek M. Bhasi, Jashwant Raj Gunasekaran, Prashanth Thinakaran, Cyan Subhra Mishra, Mahmut Taylan Kandemir, and Chita Das. 2021. Kraken: Adaptive Container Provisioning for Deploying Dynamic DAGs in Serverless Platforms. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC '21)*.

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

[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)*.

[15] Marcin Copik, Grzegorz Kwasniewski, Maciej Besta, Michal Podstawski, and Torsten Hoefler. 2021. SeBS: A Serverless Benchmark Suite for Function-as-a-Service Computing. In *Proceedings of the 22nd International Middleware Conference (Middleware '21)*.

[16] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. 2017. Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP '17)*.

[17] Nilanjan Daw, Umesh Bellur, and Purushottam Kulkarni. 2020. Xanadu: Mitigating Cascading Cold Starts in Serverless Function Chain Deployments. In *Proceedings of the 21st International Middleware Conference (Middleware '20)*.

[18] 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)*.

[19] Fission: Open source Kubernetes-native Serverless Framework. 2023. https://fission.io/.

[20] Fn Project. 2023. https://fnproject.io/.

[21] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. 2019. From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers. In *Proceedings of the 2019 USENIX Annual Technical Conference (USENIX ATC '19)*.

[22] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. 2017. Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI '17)*.

[23] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. 2020. Caladan: Mitigating Interference at Microsecond Timescales. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)*.

[24] Alexander Fuerst and Prateek Sharma. 2021. FaasCache: Keeping Serverless Computing Alive with Greedy-Dual Caching. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21)*.

[25] Google. 2023. Google Cloud Functions. https://cloud.google.com/functions.

[26] Jashwant Raj Gunasekaran, Prashanth Thinakaran, Nachiappan C. Nachiappan, Mahmut Taylan Kandemir, and Chita R. Das. 2020. Fifer: Tackling Resource Underutilization in the Serverless Era. In *Proceedings of the 21st International Middleware Conference (Middleware '20)*.

[27] Jing Guo, Zihao Chang, Sa Wang, Haiyang Ding, Yihui Feng, Liang Mao, and Yungang Bao. 2019. Who Limits the Resource Efficiency of My Datacenter: An Analysis of Alibaba Datacenter Traces. In *Proceedings of the International Symposium on Quality of Service (IWQoS '19)*.

[28] Tyler Harter, Brandon Salmon, Rose Liu, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. Slacker: Fast Distribution with Lazy Docker Containers. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST '16)*.

[29] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. Serverless Computation with OpenLambda. In *Proceedings of the 8th USENIX Conference on Hot Topics in Cloud Computing (HotCloud '16)*.

[30] IBM. 2023. IBM Cloud Functions. https://cloud.ibm.com/functions/.

[31] Zhipeng Jia and Emmett Witchel. 2021. Nightcore: Efficient and Scalable Serverless Computing for Latency-Sensitive, Interactive Microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21)*.

[32] Jiawei Jiang, Shaoduo Gan, Yue Liu, Fanlin Wang, Gustavo Alonso, Ana Klimovic, Ankit Singla, Wentao Wu, and Ce Zhang. 2021. Towards Demystifying Serverless Machine Learning Training. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21)*.

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

[34] Mahmoud Khairy, Ahmad Alawneh, Aaron Barnes, and Timothy G. Rogers. 2022. SIMR: Single Instruction Multiple Request Processing for Energy-Efficient Data Center Microservices. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO '22)*.

[35] Jeongchul Kim and Kyungyong Lee. 2019. FunctionBench: A Suite of Workloads for Serverless Cloud Function Service. In *Proceedings of the IEEE 12th International Conference on Cloud Computing (CLOUD '19)*.

[36] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. 2018. Pocket: Elastic Ephemeral Storage for Serverless Analytics. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation (OSDI '18)*.

[37] Knative. 2023. https://knative.dev/docs/.

[38] KNative Serving Activator. 2023. https://github.com/knative/serving/tree/ -main/pkg/activator.

[39] KNative Serving Autoscaler. 2023. https://github.com/knative/serving/tree/ -main/pkg/autoscaler.

[40] Swaroop Kotni, Ajay Nayak, Vinod Ganapathy, and Arkaprava Basu. 2021. Faastlane: Accelerating Function-as-a-Service Workflows. In *Proceedings of the 2021 USENIX Annual Technical Conference (USENIX ATC '21)*.

[41] Kubeless: The Kubernetes Native Serverless Framework. 2023. https://kubeless.io/.

[42] Senthil Kumar and Ajit Puthiyavettle. 2021. Architecting a Highly Available Serverless, Microservices-Based Ecommerce Site. https://aws.amazon.com/blogs/architecture/architecting-a-highly-available-serverless-microservices-based-ecommerce-site/.

[43] Huiba Li, Yifan Yuan, Rui Du, Kai Ma, Lanzheng Liu, and Windsor Hsu. 2020. DADI: Block-Level Image Service for Agile and Elastic Application Deployment. In *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*.

[44] Zijun Li, Quan Chen, Shuai Xue, Tao Ma, Yong Yang, Zhuo Song, and Minyi Guo. 2020. Amoeba: QoS-Awareness and Reduced Resource Usage of Microservices with Serverless Computing. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS '20)*.

[45] Zijun Li, Yushi Liu, Linsong Guo, Quan Chen, Jiagan Cheng, Wenli Zheng, and Minyi Guo. 2022. FaaSFlow: Enable Efficient Workflow Execution for Function-as-a-Service. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22)*.

[46] Ping-Min Lin and Alex Glikson. 2019. Mitigating Cold Starts in Serverless Platforms: A Pool-Based Approach. *CoRR* abs/1903.12221 (2019). arXiv:1903.12221 http://arxiv.org/abs/1903.12221

[47] Qixiao Liu and Zhibin Yu. 2018. The Elasticity and Plasticity in Semi-Containerized Co-Locating Cloud Workload: A View from Alibaba Trace. In *Proceedings of the 2020 ACM Symposium on Cloud Computing (SoCC '18)*.

[48] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Yu Ding, Jian He, and Chengzhong Xu. 2021. Characterizing Microservice Dependency and Performance: Alibaba Trace Analysis. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC '21)*.

[49] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Eshaan Minocha, Sameh Elnikety, Saurabh Bagchi, and Somali Chaterji. 2022. WISEFUSE: Workload Characterization and DAG Transformation for Serverless Workflows. In *Proceedings of the 2022 ACM SIGMETRICS/IFIP PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS/PERFORMANCE '22)*.

[50] Microsoft. 2023. Microsoft Azure Functions. https://azure.microsoft.com/en-gb/services/functions/.

[51] Goncalo Neves. 2017. Keeping Functions Warm - How To Fix AWS Lambda Cold Start Issues. https://www.serverless.com/blog/keep-your-lambdas-warm.

[52] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. SOCK: Rapid Task Provisioning with Serverless-Optimized Containers. In *Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC '18)*.

[53] OpenFaaS. 2023. https://docs.openfaas.com/.

[54] OpenWhisk Invoker. 2023. https://github.com/apache/openwhisk/tree/master/-core/invoker.

[55] OpenWhisk Load Balancer. 2023. https://github.com/apache/openwhisk/tree/-master/core/controller/src/main/scala/org/apache/openwhisk/core/-loadBalancer.

[56] OpenWhisk Python Runtime. 2023. https://github.com/apache/openwhisk-runtime-python.

[57] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. 2019. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI '19)*.

[58] Linux Manual Page. 2023. ld.so(8). https://man7.org/linux/man-pages/man8/ld.so.8.html.

[59] Linux Manual Page. 2023. perf-stat(1). https://man7.org/linux/man-pages/man1/perf-stat.1.html.

[60] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. 2019. Shuffling, Fast and Slow: Scalable Analytics on Serverless Infrastructure. In *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI '19)*.

[61] Henry Qin, Qian Li, Jacqueline Speiser, Peter Kraft, and John Ousterhout. 2018. Arachne: Core-Aware Thread Management. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18)*.

[62] Redis. 2023. https://redis.io/.

[63] Charles Reiss, Alexey Tumanov, Gregory R. Ganger, Randy H. Katz, and Michael A. Kozuch. 2012. Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis. In *Proceedings of the 3rd ACM Symposium on Cloud Computing (SoCC '12)*.

[64] Francisco Romero, Gohar Irfan Chaudhry, Inigo Goiri, Pragna Gopa, Paul Batum, Neeraja 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)*.

[65] Francisco Romero, Qian Li, Neeraja J. Yadwadkar, and Christos Kozyrakis. 2021. INFaaS: Automated Model-less Inference Serving. In *Proceedings of the 2021 USENIX Annual Technical Conference (USENIX ATC '21)*.

[66] Rohan Basu Roy, Tirthak Patel, and Devesh Tiwari. 2022. IceBreaker: Warming Serverless Functions Better with Heterogeneity. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22)*.

[67] 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)*.

[68] Serverless Train Ticket. 2023. https://github.com/FudanSELab/serverless-trainticket.

[69] Mohammad Shahrad, Jonathan Balkind, and David Wentzlaff. 2019. Architectural Implications of Function-as-a-Service Computing. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '19)*.

[70] Mohammad Shahrad, Rodrigo Fonseca, Inigo 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 Annual Technical Conference (USENIX ATC '20)*.

[71] Simon Shillaker and Peter Pietzuch. 2020. Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing. In *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*.

[72] Arjun Singhvi, Arjun Balasubramanian, Kevin Houck, Mohammed Danish Shaikh, Shivaram Venkataraman, and Aditya Akella. 2021. Atoll: A Scalable Low-Latency Serverless Platform. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC '21)*.

[73] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Joseph E. Gonzalez, Joseph M. Hellerstein, and Alexey Tumanov. 2020. Cloudburst: Stateful Functions-as-a-Service. *Proceedings of the VLDB Endowment* (2020).

[74] Ali Tariq, Austin Pahl, Sharat Nimmagadda, Eric Rozner, and Siddharth Lanka. 2020. Sequoia: Enabling Quality-of-Service in Serverless Computing. In *Proceedings of the 11th ACM Symposium on Cloud Computing (SoCC '20)*.

[75] 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)*.

[76] 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 *Proceedings of the 2021 USENIX Annual Technical Conference (USENIX ATC '21)*.

[77] Ao Wang, Jingyuan Zhang, Xiaolong Ma, Ali Anwar, Lukas Rupprecht, Dimitrios Skourtis, Vasily Tarasov, Feng Yan, and Yue Cheng. 2020. InfiniCache: Exploiting Ephemeral Serverless Functions to Build a Cost-Effective Memory Cache. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST '20)*.

[78] Sam Wilson and Desta Pickering. 2021. A Guide to Developing Serverless Ecommerce Workflows for Commercetools with AWS Lambda. https://aws.amazon.com/blogs/industries/a-guide-to-developing-serverless-ecommerce-workflows-for-commercetools-with-aws-lambda/.

[79] Yanan Yang, Laiping Zhao, Yiming Li, Huanyu Zhang, Jie Li, Mingyang Zhao, Xingzhen Chen, and Keqiu Li. 2022. INFless: A Native Serverless System for Low-latency, High-throughput Inference. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22)*.

[80] Tianyi Yu, Qingyuan Liu, Dong Du, Yubin Xia, Binyu Zang, Ziqian Lu, Pingchao Yang, Chenggang Qin, and Haibo Chen. 2020. Characterizing Serverless Platforms with ServerlessBench. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC '20)*.

[81] Yanqi Zhang, Íñigo Goiri, Gohar Irfan Chaudhry, Rodrigo Fonseca, Sameh Elnikety, Christina Delimitrou, and Ricardo Bianchini. 2021. Faster and Cheaper Serverless Computing on Harvested Resources. In *Proceedings of the International Symposium on Operating Systems Principles (SOSP '21)*.

[82] Laiping Zhao, Yanan Yang, Yiming Li, Xian Zhou, and Keqiu Li. 2021. Understanding, Predicting and Scheduling Serverless Workloads under Partial Interference. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '21)*.

[83] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chenjie Xu, Chao Ji, and Wenyun Zhao. 2018. Benchmarking Microservice Systems for Software Engineering Research. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings (ICSE '18)*.