Towards Efficient Processing of Latency-Sensitive Serverless DAGs at the Edge

Xiaosu Lyu
George Washington University
Washington, DC, USA

Ludmila Cherkasova
Arm Research
San Jose, CA, USA

Robert Aitken
Arm Research
San Jose, CA, USA

Gabriel Parmer
George Washington University
Washington, DC, USA

Timothy Wood
George Washington University
Washington, DC, USA

ABSTRACT

Many emerging novel applications expect "near real-time" processing and responses, which can not be guaranteed by today's Cloud and would require processing at the Edge. Serverless computing is a particularly promising architecture for edge environments since it offers to improve efficiency by precisely scaling resources to meet application needs. As the edge applications grow more complex and get composed from a subset of simpler functions or microservices, there is a need to support more complicated function topologies which can be represented as directed acyclic graphs (DAGs). However, running DAG functions on a serverless platform poses new challenges related to interconnecting, instantiating, and scheduling function sandboxes. In this paper[1], we explore how Sledge, a Wasm-based serverless runtime, can be extended to support DAG functions. Sledge's unique design allows for extremely lightweight sandbox instantiation — a new sandbox can be started for each function invocation in under 30μsec — which mitigates the cold start problems that can be especially detrimental to DAGs. Rather than relying on expensive coordination via shared storage, the enhanced Sledge framework provides a fast memory communication channel to propagate data through the DAG. We consider the DAGs with service level objectives, defined by their execution deadlines. To ensure the DAGs meet their performance requirements, we consider, analyze, and compare two deadline-aware pluggable schedulers (that we implemented in Sledge) on a variety of realistic workloads.

CCS CONCEPTS

• **Computing methodologies** → **Distributed computing methodologies**; • **Computer systems organization** → *Real-time system architecture.*

[1]This work was mostly completed during X. Lyu's summer internship in 2021 at Arm Research. G. Parmer and T. Wood are partially supported by SRC GRC tasks 3046.001 and 2911.001, and NSF grants CNS-1815690 and CNS-1837382.

EdgeSys '22, April 5–8, 2022, RENNES, France

ACM ISBN 978-1-4503-9253-2/22/04…$15.00
https://doi.org/10.1145/3517206.3526274

KEYWORDS

Edge computing, WebAssembly, serverless, DAGs, SLOs
ACM Reference Format:
Xiaosu Lyu, Ludmila Cherkasova, Robert Aitken, Gabriel Parmer, and Timothy Wood. 2022. Towards Efficient Processing of Latency-Sensitive Serverless DAGs at the Edge . In *5th International Workshop on Edge Systems, Analytics and Networking (EdgeSys '22), April 5–8, 2022, RENNES, France.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3517206.3526274

1 INTRODUCTION

Getting high value from Industrial IoT and next generation technologies (such as smart manufacturing, autonomous vehicles, AR/VR, etc.) requires new approaches along the entire data processing pipeline: how data is transferred, processed, ingested, and acted upon. These emerging near real-time systems and applications might demand latency in the tens of milliseconds or less [30], which cannot be offered by today's cloud services. The reliance on cloud-based technologies for time-critical services is challenging due to high networking latencies and performance uncertainties implicit in today's cloud models. Such latency-sensitive applications have to be processed at the Edge. Edge computing can offer clear advantages when dealing with low latency, connectivity, security or privacy as well as when transmitted data volumes can be an issue.

"Serverless" computing, which is also known as Function-as-a-Service (FaaS), offers a new execution model, where a user can upload and execute a small application (micro-service) without handling operational issues around server provisioning, resource management, and capacity scaling. Current FaaS solutions support stateless functions that typically require minimal I/O and communications. All the major cloud providers offer serverless solutions [2, 11, 14, 22]. While implementation details differ, most FaaS offerings utilize some sandboxing environment (like VMs or containers) for executing serverless functions. These frameworks are somewhat heavy-weight for operating at the Edge due to their large memory footprint and high startup time (cold start) [9, 29]. Startup delay varies across different platforms from 125ms for AWS Lambda [7] to 1sec for Microsoft Azure. While one can improve "cold start" by caching and reusing containers or minimizing the invocation time via snapshotting [6, 27], this does not change the solution memory footprint. Other approaches to reduce FaaS resource footprint and invocation time at the Edge are through light-weight isolation runtimes based on WebAssembly [8, 13, 24, 26] and unikernels [17].

Among the new recent trends is the use of serverless computing for complex data processing pipelines, e.g., for video/image

analytics or ML workflows. Under this model, the data processing application is defined as a DAG (Direct Acyclic Graph), where each node represents an invocation of a different serverless function. To meet user demands, the major serverless providers recently introduced support for serverless function workflow composition, such as AWS Step Functions [3], Azure Durable Functions [4], and Google Cloud Composer [10]. Since the existing serverless platforms are stateless the data exchange between the functions in the DAG requires saving and loading this data through remote storage (e.g., S3) as shown in Figure 1. It was reported in [20] that passing data through remote storage could consume over 75% of the function execution time. Thus exchanging intermediate data between the functions is *a major challenge* in the cloud serverless workflows.

The critical difference and problem when designing solutions for the Edge, compared to the Cloud, is that the Edge represents a resource-constrained environment, and therefore, some traditional cloud-based platforms might need to be replaced by a new leaner and lighter solution for the Edge. While the elimination of long network delays when accessing the Edge makes it ideal for low latency services such as cyber physical and AR/VR systems, serverless cold start delays could eliminate this benefit. For serverless platforms to meet the requirements of Edge computing, they must support complex application topologies such as DAGs, while using minimal resources and providing predictable bounds on response time. Unfortunately, to our knowledge no serverless platform today can provide these features.

Our Contributions: In this paper, we extend the open-source Wasm-based serverless framework Sledge[2][8]:

- We explore how Sledge can support efficient processing of serverless DAGs, in particular by replacing the expensive intermediate data coordination and transfer via shared storage with *a fast memory communication channel* for propagating the intermediate data through a given DAG.
- We show that the overheads of sandbox and communication channel creation can be orders of magnitude faster than VMs or containers: only around 25 μsec due to our use of lightweight Wasm sandboxes.
- We investigate how to support DAGs with service level objectives (SLOs) defined by execution deadlines and offer a set of "pluggable" schedulers, such as (i) Earliest Deadline First (EDF), driven by the DAGs deadlines, and (ii) Shortest Remaining Slack First (SRSF), where the DAG remaining execution time ("slack") defines the priority of its execution.
- We perform a detailed sensitivity analysis of EDF vs SRSF for a variety of workloads and deadlines. Overall, the efficiency of both policies are close. However, EDF miss rates are smaller for "short" execution DAGs (with "earlier" deadlines) and worse for "longer" executing DAGs, while SRSF miss rates are similar across different DAGs classes due to more "fair" slack-based priorities based on the DAGs slack (remaining execution time).

2 BACKGROUND AND RELATED WORK

Serverless Execution Frameworks. Since the appearance of Amazon Lambda in 2014, all the major cloud providers have designed

[2]https://github.com/gwsystems/sledge-serverless-framework

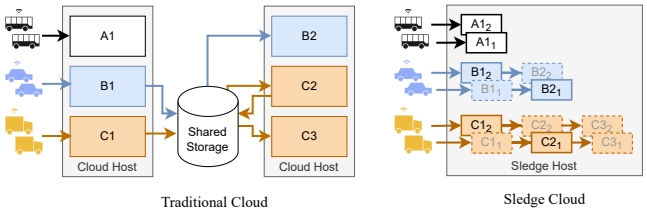

Figure 1: Traditional cloud serverless platforms use shared function instances for multiple clients of the same type, and propagate data between functions via shared storage. Sledge allows a dense deployment of functions instantiated on demand for each user, with efficient memory-based communication.

and implemented their serverless solutions. These solutions offer to user functions strong isolation guarantees provided by VMs or Containers platforms. However, these frameworks are heavy-weight due to their large memory footprint and high startup time. New low-overhead sandboxing mechanism were released by cloud providers such as Google gVisor and Amazon Firecracker. For example, gVisor has a footprint of 15MB and will boot up in 150ms[15]. These metrics are important and they do reflect that these frameworks might still be not "light-enough" for using at the Edge.

Light-weight Isolation Platforms for the Edge. Recently, WebAssembly (Wasm) [12] with its light-weight memory sanboxing has emerged as a promising approach for supporting serverless at the Edge as shown by commercial products introduced in 2019 from Cloudflare [26, 28]. Fastly's native WebAssembly compiler and runtime Lucet [19] can instantiate Wasm modules in under 50 μsec. Similarly, in the Sledge framework [8], the average function startup time is around 30 μsec. The single-process Sledge runtime binary size is 359 KB, and it enables functions to share the library dependencies, while providing a strong spatial and temporal isolation for multi-tenant functions executions.

Microservices and Stateful Applications Issues. Microservices offer a new appealing model to ease the application development. Serverless computing is also embracing this model, which becomes increasingly popular for defining complex workflows. However, serverless functions are stateless. Therefore, support for exchanging intermediate data between the functions (e.g., via S3 remote storage) could lead to a significant performance overhead in the cloud serverless workflows (up to 75% of overall latency) [20].

In SAND [1], the authors achieve good performance by using application sandboxing (i.e., running the functions of the same application DAG as processes within the same container). Similar ideas are pursued in the Nightcore [16] and Faasm [24] frameworks. Since Faasm runtime manages and isolates functions by compiling them to WebAssembly, it is closest to our Sledge-based solution. Faasm focuses on shared memory abstractions between sandboxes that require changes to function implementations, and manages system resources by relying on Linux utilities, while Sledge runtime gets additional execution efficiency (and user-level function scheduling) due to kernel bypass.

Latency Critical Applications and SLOs. Many user-facing or interactive applications have stringent SLOs, where a service has to be delivered within a specified "soft" deadline. Two recently introduced solutions Atoll [25] and Kraken [5] pursue these objectives. Atoll framework offers redesigned control and data planes management and utilizes Shortest Remaining Slack First (SRSF)

scheduling to demonstrate significantly improved results compared to state-of-the-art alternatives. Kraken extends these results for a class of applications defined by Dynamic DAGs. In our work, we analyze the scheduling efficiency with SRSF and compare its performance with a simpler EDF request scheduling.

3 DESIGN AND IMPLEMENTATION

3.1 Sledge Framework Background

Sledge is a lightweight, WebAssembly based serverless platform [8]. As shown in Figure 2, Sledge is a single process with multiple worker threads, each implemented as a Linux `pthread`. Each worker thread is pinned to a CPU core to reduce Linux context-switch overhead, and each provides user-level scheduling of WebAssembly-based serverless function sandboxes. A single "Listener" thread accepts client requests using Linux kernel networking, and puts those requests into a global runqueue. At this point, the only memory allocated for the request is some basic meta data, illustrated by the Pending Sandboxes in Figure 2. Here each sandbox is labeled with its Chain type, the stage in the function chain, and the user it is allocated to (e.g., $B2_1$ is a sandbox for service B's stage 2, instantiated for user 1). The diagram also indicates the deadline, D, for each task, which is used by the scheduler described below. The A, B, C chains match those from Figure 1.

Once a request is taken from the global queue, a WebAssembly-based sandbox is instantiated to handle that specific request. This can achieve high resource utilization without introducing too much latency because Sledge provides a micro-second level cold-start overhead. Sledge relies on its aWsm compiler [9] to generate safe sandbox code and produces a memory layout that ensures software fault isolation, i.e., the code inside a function unable to access memory addresses or jump to instructions outside its sandbox range.

Each worker thread executes sandbox functions by competing to get requests from the global queue into its local runqueue. Thus, each worker can execute requests in its local queue, or get a new request. Checking for a new request happens: 1) when the local queue is empty, 2) when a sandbox is blocked due to IO, or 3) when a timer tick interrupts the worker each quantum (sent using a periodic `SIGALRM` signal that occurs every 5 milliseconds).

To scale and efficiently execute functions, Sledge uses kernel-bypass, user-level schedulers which are triggered when functions terminate, and at each quantum. Each worker thread schedules its local queue and acquires based on a configurable scheduler policy. When the scheduler is triggered, it will decide whether to get a new request from the global queue, or to continue executing the active request by comparing the task's priority in the local queue and global queue.

3.2 DAG/Chain Functions Support

To enhance Sledge for supporting DAG or Chain functions requires us to resolve two primary questions: 1) how/when should the different stages of a complex service be instantiated, and 2) how should data be communicated between stages.

For typical serverless platforms built using containers or virtual machines, cold start delays can be particularly damaging for DAG functions since the startup cost can be incurred at each stage of the pipeline if the system is not able to preemptively warm the appropriate functions. Fortunately, Sledge's use of lightweight sandboxes

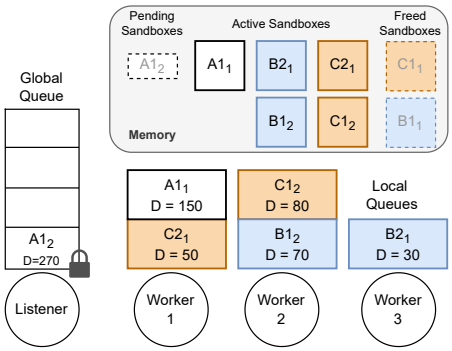

Figure 2: Sledge Architecture.

sidesteps this challenge by reducing the cost of starting a new function to microsecond scale instead of the normal 100 milliseconds or more needed to start a VM or container [7, 21]. Thus we extend Sledge to support function chains by deploying each function in the DAG as a separate sandbox, and enqueuing the next stage of the DAG into the global queue when the prior one finishes. This means that for a given DAG, only the actively executing stages will be resident in memory at one time. This is illustrated in Figure 2, where pending sandboxes for requests still in the global queue require only a small amount of meta data ($A1_2$), runnable sandboxes are active in memory ($A1_1$, $B1_2$, $B2_1$, $C1_2$, and $C2_1$), and sandboxes that have finished ($C1_1$ and $B1_1$) or have not yet started ($B2_2$, $C3_1$, $C2_2$, and $C3_2$) incur zero memory cost.

We choose do not put the next request of the chain to the local runqueue for two reasons. First, some worker threads' runqueues might be very long, but others might be short. Putting the subsequent requests to the local runqueue directly will exacerbate this imbalance, reducing performance. The second reason is to maximize parallelism. When a sandbox finishes execution and adds the next stage to the queue, there is still some cleanup work to do for the old sandbox. If the subsequent request is processed by the same worker thread, it must wait for the cleanup to complete. To both distribute the load more evenly and maximize concurrency, we put all subsequent requests to the global queue instead of the local runqueue.

The next issue for the DAG function support is how to efficiently share the intermediate state and output between the functions. Current approaches, using either a remote storage system like S3 or local storage, have high latency overheads due to network transfers and slow disk IO. The alternative approach is to use message queues (e.g., Apache Kafka or KubeMQ), but these frameworks might still be too overhead-heavy for resource-constrained edge nodes.

Our solution for this problem is to do message passing based on a memory communication channel. We allocate a temporary memory space for each function's output and input. In order to ensure strong isolation, we rely on the Sledge runtime to copy the output of a prior function into the input buffer of the next function. This minimizes the data transfer cost and avoids all locks, but still retains the strong sandbox isolation guarantees promised by Sledge.

3.3 Pluggable Function Scheduling Algorithms

Sledge avoids the Linux kernel scheduler and takes a full control of serverless function executions and their management, including function profiling, scheduling and resource allocation. This design

opens up a set of interesting opportunities for customizable SLO-driven performance management of users' functions and DAGs of functions based on their deadlines. To achieve this goal, we offer a set of "pluggable" schedulers, such as EDF and SRSF.

EDF is the *earliest deadline first* scheduling algorithm. For DAG functions, each sub-function in the DAG shares the same deadline. EDF ordering is based on the tasks' deadlines, i.e., it does not consider the functions' execution times. A function with the earliest deadline has the highest priority to be executed. Recall from the Sledge architecture that workers can either run the lowest deadline task from their runqueue, or they can take a task from the global queue if it has a lower deadline; tasks cannot be rebalanced between workers. Thus in Figure 2, once Worker 1 finishes task $C2_1$, it will process task $A1_1$, even though its deadline, $D = 150$, may be further away than a task in Worker 2's queue, such as $C1_2$ with deadline $D = 80$. Only if a task with a shorter deadline arrives in the global queue will $A1_1$ be preempted. While this architecture can lead to suboptimal scheduling decisions, it eliminates the need to have locks on the local queues and prevents overheads related to task migration, which is a good trade-off for two reasons. First, as the number of cores increases, work more easily is balanced across core. Second, serverless employs *transient* computations in response to a client's invocation, thus long-term work imbalances across cores are unlikely.

SRSF is the *shortest remaining slack first* scheduling algorithm. Here "slack" represents the amount of time a task can safely be queued without missing its deadline. If the task deadline is D, and the task execution time is E, then its *initial* remaining slack $R = D - E$. If the task is queued for Q units waiting to be executed, its remaining slack will be reduced by Q. For an executing task, its remaining slack will not change. A task with the smallest slack has the highest urgency. With SRSF, a worker thread always picks the task with the smallest remaining slack to execute from either its local runqueue or the global queue.

SRSF algorithm requires profiling information about the functions' execution times. By collecting the execution times online, we obtain for each function its execution time distribution. Depending on the needs of the system, an appropriate percentile can be used from the distribution, e.g., 50th percentile leads to a more lax scheduling, while an estimate based on the response time tail leads to a more conservative scheduler.

Another important SRSF related issue is its efficient implementation. It relies on the priority queue, where the function's remaining slack defines its priority. As we mentioned earlier, the function queuing time decreases the slack. Therefore, a new request insertion might require $O(n + log(n))$, where $O(n)$ is needed for updating all tasks' remaining slack in the queue and $O(log(n))$ is for inserting a new task into the queue. In comparison, the overhead of *insert* operation in EDF is $O(log(n))$. To avoid this extra overhead, our SRSF implementation maintains an additional timestamp for when a task's slack was last updated, and we proactively update the slack value whenever a task starts to run, rather than waiting until a new task arrives. In this way we are able to reduce the time complexity to match EDF.

4 IMPLEMENTATION & EVALUATION

We implemented our design of Sledge extension in C (4K LOC). Our current implementation[3] supports linear chain functions. We leave the more complex DAG functions to be our future work. The DAG functions' structure and deadline settings can be defined in a configuration file.

We perform our experiments and evaluation with the servers from the CloudLab testbed. We use one node to represent the Edge environment (running Sledge) and utilize the second node as a workload generator. The nodes run on the Massachusetts site and are type rs620, with **16** cores, **264GB** memory, and **10Gbps** NIC.

4.1 Overhead Assessment

We evaluate the sandbox initialization cost by measuring the overhead for a chain function with a sequence of simple no-op (null) functions. By increasing the number of no-op functions in the chain, we can observe the cost to instantiate and schedule the functions. Table 1 shows that starting a single function takes only 12 microseconds, and while the cost rises for longer chains, Sledge can start five functions about 1000x faster than Amazon Firecracker (which is used to run AWS Lambda serverless functions) can start a single microVM (125 milliseconds [7]). We also show the end to end latency seen by a client issuing requests to the function chain, which achieves sub millisecond response times in all cases.

	1 no-op	2 no-op	3 no-op	4 no-op	5 no-op
Sandbox Initiation	12us	50us	86us	120us	159us
E2E Latency	0.40ms	0.49ms	0.50ms	0.70ms	0.80ms

Table 1: Average Cold-Start And Latency of no-op Function Chains.

Note the slight increase in the initiation time for the chain of sandboxes. This includes additional time for setting up the memory communication channel between the sandboxes, which takes approximately 25 microsecond for this operation per sandbox. These overheads for wasm box initiation and memory channel communication setup stay consistent in our measurements for the chain of 5 no-op functions as shown in Table 1.

Next, we evaluate the overhead incurred when a monolithic function is broken into components. In order to arbitrarily subdivide a function, we use a series of ten computationally intensive Fibonacci number calculations to represent the workload. We compare running all of the calculations in a single sandbox versus splitting them into ten different sandboxes. We find that on average, the single monolithic function takes 24.5 milliseconds per request, while the chain of ten functions takes 25.9 milliseconds. Thus the total cost of instantiating nine additional sandboxes, scheduling each of them, and communicating data between them incurs less than 1.5 milliseconds (6%) overhead.

4.2 SRSF vs EDF Performance Comparison

Experimental Workload. While EDF is more popular in real-time contexts, SRSF has been used in [25] to provide timely serverless computation. To study both, we perform the analysis and performance evaluation with a complex *image classification application*. It consists of three sub-functions chained together:

[3] https://github.com/lyuxiaosu/sledge-serverless-framework/tree/linear_chain_srsf

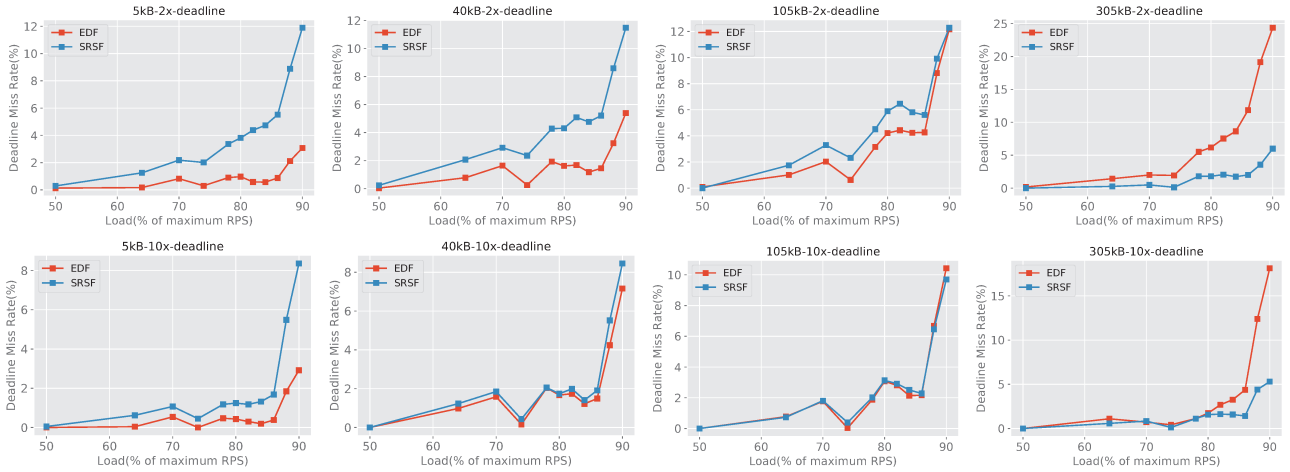

Figure 3: Deadline Miss Rate for DAGs execution with Different Inputs and 2x,10x Deadline Setting.

(1) *Image Resize:* resizes a jpg format image to half its original size and then outputs a png format image.
(2) *Format Transformation:* transforms an image in png format to bmp format.
(3) *Image Classification:* reads a bmp format image and classifies it (by executing a classifier of 10 classes and writing the number associated with the resulting class to stdout).

Diverse Workload Mix. To simulate a realistic workload with different function profiles [23] for processing time and communication costs, we use four sizes of input images for our image classification application. The four input sizes are 5KB, 40KB, 105KB, and 305KB. In an edge environment, this could represent data streams from IoT devices with different fidelity sensors. Table 2 shows the execution time distribution profile of our four experimental workloads (based on 1000 execution runs of each workload). The maximum number of requests per second (RPS), shown in the last column for each sub-workload, represents the share of each class under 100% load.

DAG Input Size KB	min ms	50% ms	80% ms	95% ms	Workload Percentage	Max RPS
5 KB.jpg	7.47	8.17	8.60	9.16	60%	430
40 KB.jpg	21.67	23.91	24.66	25.49	25%	179
105 KB.jpg	35.48	39.28	40.31	41.24	10%	72
305 KB.jpg	88.19	96.38	98.02	99.58	5%	36

Table 2: Execution time distribution profile of the four experimental workloads under study.

The difference in function execution times across all four workload classes is relatively narrow. Therefore, we use 50% percentile as an estimate of the function execution time. This metric and estimate is needed for defining the maximum applied load in our experiments as well as for estimating the function execution time used in the SRSF scheduling policy.

Experimental Workload Mix Composition. Commercial serverless workload studies [23] demonstrate the dominance of small and medium function sizes. We use this guidance for percentages in our workload mix as shown in Table 2. Therefore, up to 95% of our DAGs executions represent small and medium duration functions, with remaining 5% of the requests being longer duration DAGs executions.

Mixed Deadline Settings for DAGs. To analyze the impact of different deadline settings in our experimental workload, we partition the workload so that half of the requests have a tight deadline (two times the median execution cost) and half of the requests have a loose deadline (ten times the median execution cost).

Workload Generator. We modified `Loadtest` [18] - an open-loop workload generator to let the requests follow a Poisson distribution. To replay the same workload with different scheduling algorithms, we used the same random sequence seeds to generate a repeatable workload. Since we have four workloads with two deadline settings for each, we started eight `Loadtest` instances in parallel, with each `Loadtest` sending the *image classification application* chain with one specified file size and deadline setting. The generators of each file size send a workload percentage as indicated in Table 2. We gradually increase the client load as a percentage of the maximum RPS (requests per second) supported by the system and measure the miss rate at each load level.

Results: SRSF vs EDF Comparison. Figure 3 shows the deadline miss rates for DAG executions with different input sizes (5KB, 40KB, 105KB, 305KB) and deadline settings (2x in top row, 10x in bottom). Table 3 shows the details for the deadline miss rates under 90% load.

	5 KB	40 KB	105 KB	305 KB
SRSF-2X	11.9%	11.5%	12.3%	6.0%
EDF-2X	3.0%	5.4%	12.2%	24.4%
SRSF-10X	8.4%	8.4%	9.7%	5.3%
EDF-10X	2.9%	7.2%	10.4%	18.1%

Table 3: 2X and 10X Deadline Miss Rates Under 90% Load

These results are interesting: they show mixed performance:

- EDF achieves better deadline miss rates than SRSF for DAGs with smaller size inputs of 5KB and 40KB;
- EDF and SRSF performance gets very close for DAGs with inputs of 105KB;
- SRSF achieves better deadline miss rates than EDF for DAGs with large 305 KB file inputs.

This behavior is due to the smaller request sizes having lower deadlines (as deadlines are proportional to execution time), which

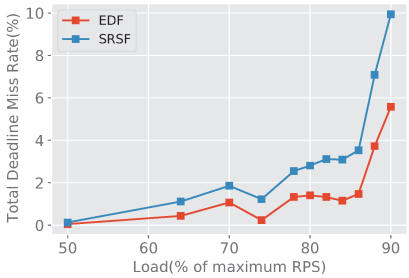

Figure 4: Total Deadline Miss Rate

EDF will prioritize. By comparing the deadline miss rate in the top section to the bottom section, we find that Sledge shows a consistently lower miss rate for requests with the 10X deadline, since the looser deadline is easier to meet.

The lower miss rate of EDF under small file sizes means that a resource constrained Edge server could handle a larger request rate while still meeting a deadline target, while the same is true for SRSF for handling larger files. If a system had a target miss deadline rate of 2%, EDF could handle a mixed workload with approximately 378 5KB req/sec, whereas SRSF only handles 300 req/sec. For the larger 305KB files, SRSF can maintain a rate of 28 req/sec with a miss rate under 2%, whereas EDF only reaches 25 req/sec.

When we consider the workloads as a whole in Figure 4, we find that EDF consistently achieves a lower total miss-deadline rate. EDF can maintain 617 total req/sec (86% load) while meeting a 2% miss rate, whereas SRSF can only achieve 502 req/sec (70% load) for the same target. Thus in total, EDF is capable of handling about 23% more requests per second without violating deadlines. It should be noted that our workload is skewed towards smaller requests with shorter deadlines (60% are 5KB while only 5% are 305KB), which benefits EDF. We expect that this will commonly be the case in a real environment, but our results suggest that for workloads with an opposite skew, or where heavy requests have greater priority, a different scheduling algorithm might be preferred.

5 CONCLUSION AND FUTURE WORK

Edge applications deployed as DAGs or chains create significant challenges for serverless platforms due to the high cost of cold start delays and communication overheads. Our work has shown how the lightweight sandboxes used by Sledge provide an ideal environment for deploying chains of functions. Instantiating a chain of five functions can be done in as little as 159 *microseconds*, multiple orders of magnitude faster than existing container and micro VM approaches. By allowing direct communication between functions – while still retaining strong isolation guarantees – Sledge minimizes the cost of data transfer between functions. Finally, our evaluation of different scheduling algorithms shows that even a simple earliest-deadline first policy can support a low miss rate at high utilization, although it may lead to longer delays for larger requests.

In our future work, we are continuing to expand Sledge with support for more complex DAG functions and exploring how the scheduling and admission control systems can be improved to provide stronger deadline guarantees. We are evaluating how machine learning models can be used to predict the execution cost of functions within a DAG in order to make better scheduling and resource allocation decisions.

REFERENCES

[1] I. Akkus, R. Chen, I. Rimac, M. Stein, K. Satzke, A. Beck, P. Aditya, and V. Hilt. 2018. SAND: Towards High-Performance Serverless Computing. In *Proc. of the 2018 Usenix Annual Technical Conference.*

[2] AWS Lambda 2014. https://aws.amazon.com/lambda/.

[3] AWS Step Functions 2020. https://amazon.com/step-functions/latest/dg/concepts-amazon-states-language.html.

[4] Azure Durable functions 2020. https://docs.microsoft.com/en-us/azure/azure-functions/durable/.

[5] V. M. Bhasi, J. Gunasekaran, P. Thinakaran, C. Mishra, M. Kandemir, and C. Das. 2021. Kraken: Adaptive Container Provisioning for Deploying Dynamic DAGs in Serverless Platforms. In *Proc. of the ACM Symp. on Cloud Computing (SoCC'21).*

[6] D. Du, T. Yu, Y. Xia, B. Zang, G. Yan, C. Qin, Q. Wu, and H. Chen. 2020. Catalyzer: Sub-Millisecond Startup for Serverless Computing with Initialization-Less Booting. In *Proc. of the 25th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems.*

[7] Firecracker-microVM 2019. https://github.com/firecracker-microvm/firecracker/blob/master/docs/snapshotting/snapshot-support.md.

[8] P. Gadepalli, S. McBride, G. Peach, L. Cherkasova, and G. Parmer. 2020. Sledge: A Serverless-First, Light-Weight Wasm Runtime for the Edge. In *Proc. of the 21st Intl. Middleware Conference.*

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

[10] Google Cloud Composer 2020. https://cloud.google.com/composer.

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

[12] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J.F Bastien. 2017. Bringing the Web Up to Speed with WebAssembly. In *Proc. of the 38th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI '17).*

[13] A. Hall and U. Ramachandran. 2019. An Execution Model for Serverless Functions at the Edge. In *Proc. of the Intl. Conf. on Internet of Things Design and Implementation (IoTDI '19).*

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

[15] Interview: Google gVisor and the Challenge of Securing Multitenant Containers 2018. https://thenewstack.io/interview-google-gvisor-and-the-challenge-of-securing-multitenant-containers.

[16] Zhipeng Jia and Emmett Witchel. 2021. Nightcore: Efficient and Scalable Serverless Computing for Latency-Sensitive, Interactive Microservices. In *Proc. of the 26th ACM Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2021).*

[17] Ricardo Koller and Dan Williams. 2017. Will Serverless End the Dominance of Linux in the Cloud? *(HotOS '17).*

[18] loadtest 2020. https://www.npmjs.com/package/loadtest.

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

[20] A. Mahgoub, K. Shankar, S. Mitra, A. Klimovic, S. Chaterji, and S. Bagchi. 2021. SONIC: Application-aware Data Passing for Chained Serverless Applications. In *2021 USENIX Annual Technical Conference.*

[21] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. 2017. My VM is Lighter (and Safer) Than Your Container. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP).*

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

[23] M. Shahrad, R. Fonseca, I. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini. 2020. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In *Proc. of USENIX ATC.*

[24] S. Shillaker and P. Pietzuch. 2020. Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing. In *Proc. of USENIX Annual Technical Conference.*

[25] A. Singhvi, A. Balasubramanian, K. Houck, M. Shaikh, S. Venkataraman, and A. Akella. 2021. Atoll: A Scalable Low-Latency Serverless Platform. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC'21).*

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

[27] D. Ustiugov, P. Petrov, M. Kogias, E. Bugnion, and B. Grot. 2021. Benchmarking, Analysis, and Optimization of Serverless Function Snapshots. In *Proc. of the 26th ACM Intl. Conf. on Architectural Support for Programming Languages and Operating Systems.*

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

[29] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift. 2018. Peeking Behind the Curtains of Serverless Platforms. In *Proc. of USENIX Annual Technical Conference.*

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