Rethinking Deployment for Serverless Functions: A Performance-first Perspective

Yiming Li
College of Intelligence & Computing (CIC), Tianjin University, Tianjin Key Lab. of Advanced Networking
Tianjin, China
l_ym@tju.edu.cn

Laiping Zhao*
CIC, Tianjin University, TANKLAB
Tianjin, China
laiping@tju.edu.cn

Yanan Yang
CIC, Tianjin University, TANKLAB
Tianjin, China
ynyang@tju.edu.cn

Wenyu Qu
CIC, Tianjin University, TANKLAB
Tianjin, China
wenyu.qu@tju.edu.cn

ABSTRACT

Serverless computing commonly adopts strong isolation mechanisms for deploying functions, which may bring significant performance overhead because each function needs to run in a completely new environment (i.e., the "one-to-one" model). To accelerate the function computation, prior work has proposed using sandbox sharing to reduce the overhead, i.e., the "many-to-one" model. Nonetheless, either process-based true parallelism or thread-based pseudo-parallelism still causes high latency, preventing its adaptation for latency-sensitive web services.

To achieve optimal performance and resource efficiency for serverless workflow, we argue an "*m-to-n*" deployment model that manipulates multiple granularities of computing abstractions such as processes, threads, and sandboxes to amortize overhead. We propose *wrap*, a new deployment abstraction that balances the trade-offs between interaction overhead, startup overhead and function execution. We further design *Chiron*, a *wrap*-based deployment manager that can automatically perform the orchestration of multiple computing abstractions based on performance prioritization. Our comprehensive evaluation indicates that *Chiron* outperforms state-of-the-art systems by 1.3×-21.8× on system throughput.

CCS CONCEPTS

• **Computer systems organization → Cloud computing**.

KEYWORDS

Serverless Workflows, Deployment Model, Graph Partition

ACM Reference Format:
Yiming Li, Laiping Zhao, Yanan Yang, and Wenyu Qu. 2023. Rethinking Deployment for Serverless Functions: A Performance-first Perspective. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '23), November 12–17, 2023, Denver, CO, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3581784.3613211

*Corresponding author: laiping@tju.edu.cn.

1 INTRODUCTION

Serverless computing has gained popularity as a means of developing intricate, parallel applications by interconnecting fine-grained functions into workflows [4, 32, 37, 64]. Typically, these workflows consist of data dependencies and parallel branches and can be structured as directed acyclic graphs (DAG). The orchestration of serverless workflows has been facilitated by both commercial clouds (e.g., AWS Step Functions (ASF) [54], Microsoft Durable Functions [39], and Alibaba Serverless Workflow [5]), as well as open-source systems (e.g., OpenWhisk Composers [20] and Fission Workflows [47]).

During the deployment phase of a workflow, it is customary to deploy functions within isolated sandboxes, following the *"one-to-one model"* in Figure 1. Upon receiving a request, the platform retrieves the relevant image and initiates a sandbox (e.g., microVM [55] or container) to execute the designated entry function. Subsequent functions along the call path are further instantiated one after another, each within a distinct sandbox. The *"one-to-one" model*, while offering an intricate scaling capability that promotes cost and resource efficiency, negatively impacts the end-to-end performance of serverless workflows. First, the initialization of a sandbox, which can range from hundreds of milliseconds to several seconds, can dominate the overall latency of serverless workflows [9, 22, 42, 50], e.g., starting a Hello-world Python container takes 167 ms [63]. Second, the intermediate data transfer incurs significant cost owing to the stateless formulation of functions [27, 32, 37]. This often necessitates a reliance on third-party storage services such as Amazon S3 [53] for intermediate data transfer. While the overhead generated by the cold startup of the sandbox can be circumvented through pre-warming or mitigated through startup acceleration [9, 61], intermediate data transfer can still constitute up to 95% of end-to-end latency [30]. In addition, memory redundancy due to duplicated runtimes and libraries across sandboxes can lead to excessive memory consumption [11, 31], further constraining the maximum number of instances on a machine.

Despite the strong isolation provided by the *"one-to-one" model*, it severely constrains the serverless transformation of latency sensitive applications. For instance, interactive web services like social

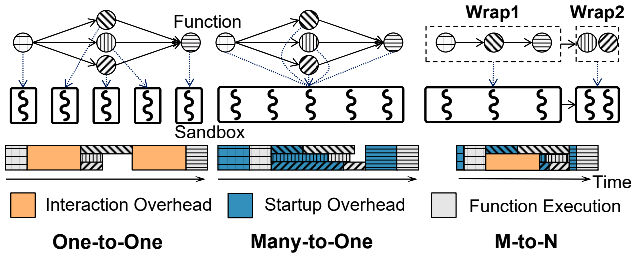

Figure 1: Schematic overview of deployment model in serverless workflow.

networks and web search typically require < 100 milliseconds latency targets [23, 27, 59, 60], which is far less than the > 1 seconds overhead resulting from cold start and interaction [9, 27, 32, 42, 63]. *Although the "isolation-first" deployment mechanism is critical in the public cloud, there exist scenarios that isolation is not that important, e.g., private cloud, development and testing environments, and non-sensitive workloads.* Even in the public cloud, it is possible to enhance the performance while reducing isolation level between functions of the same tenant.

Prior studies [1, 4, 12, 30, 37] have explored the *"many-to-one" deployment model* (Figure 1), which expands the deployment granularity by reusing a single sandbox between functions within a workflow. This model is advantageous as it mitigates excessive overheads in repeatedly initializing sandboxes and language runtimes, and enables direct communication through shared memory for reducing intermediate data transfer costs. Despite these benefits, our analysis indicates that the *"many-to-one" model* is still far from optimal due to the high startup overhead of processes coded in some popular programming languages. For example, the startup time of a Python process can be 10× greater than the execution time of some short-running functions (Figure 5). For parallel functions in a workflow that are forked one after another in a sequence (instead of at the same time), the startup time of a process, including waiting and fork syscall, may exceed the sum of the cold start time and the interaction time in the *"one-to-one" model*.

In this paper, we advocate for an "*m-to-n*" deployment model (Figure 1) that prioritizes performance, deploying *m* functions into *n* sandboxes for execution, where *m* is greater than or equal to *n*. Specifically, we partition a serverless workflow into *n* parts and deploy each of them into separate sandboxes. This model enables optimal end-to-end performance by grouping functions with minimal startup overhead and data transfer overhead together. Hence, we introduce *wrap*, a novel abstraction for guiding deployments. A *wrap* consists of a subset of functions within a workflow and serves as the fundamental unit for allocating a sandbox. The functions of a workflow may be partitioned into multiple *wraps*.

However, finding the optimal deployment granularities of *wraps* poses a significant challenge. The determination of which functions should be grouped within the same *wrap*, and the management of functions within the *wrap*, require careful consideration. *Wrap* may execute a function by either cloning a thread from an existing process or forking a new process. While thread reduces startup latency by 96% compared to process (Figure 5), concurrent execution of multiple threads is disallowed in the popular runtimes of serverless such as Python and Node.js [3, 14, 15], resulting in sub-optimal execution performance (e.g., 2× latency in Figure 6). Thus, identifying the functions partitioning and execution mode for each function, that can ensure optimal performance and resource efficiency based on workload heterogeneity in a serverless workflow, remains an immensely challenging task.

To overcome these challenges, we design *Chiron*, a system that leverages *wrap* abstraction with combined processes and threads. It can achieve 19.5× and 7.6× higher throughput on average than the *"one-to-one" model* and *"many-to-one" model*, respectively. To efficiently explore optimal *wraps* for serverless workflows, we design *PGP*, a **p**rediction-based **g**raph **p**artitioning algorithm for determining function sets within each *wrap* and execution mode for each function. *PGP* provides an accurate end-to-end latency predictor, as well as a heuristic algorithm that addresses two fundamental optimization related to performance and resource efficiency.

Contributions. We highlight the contributions as follows:

- **Problem**: An analysis of trade-off between performance and overhead in existing deployment models, and an investigation of weakness in their design.
- **Wrap**: A novel abstraction for the *"m-to-n"* deployment model, aimed at optimizing performance and resource efficiency by function partition and function execution mode selection.
- **PGP**: A prediction-based graph partitioning algorithm for exploring the optimal deployment granularity of *wrap*.
- **Chiron**: A deployment system that implements *wrap* and *PGP* atop OpenFaaS with combined processes and threads, is developed.
- **Evaluation**: Experimental results on various serverless workflows demonstrate the performance and resource efficiency of the *"m-to-n"* deployment model and system.

2 BACKGROUND & MOTIVATION

In this section, we highlight the inefficiency of existing deployment models in serverless workflows and motivate the new abstraction.

2.1 Serverless Workflow & Deployment Model

Serverless computing enables the "composable development capability" for cloud applications. Applications can be built using functions as a service (FaaS) and these functions can be combined to form larger workflows. A workflow comprises a sequence of execution stages, wherein each stage exhibits either a *sequential* or *parallel* pattern.

Deployment Model. The *"one-to-one" deployment model*, spawning individual sandboxes to execute each function in isolation, is the dominant paradigm in current platforms [5, 20, 39, 47, 54]. However, it suffers from sub-optimal performance efficiency due to the cascading startup overhead of sandboxes and interaction overhead between functions through network [8, 30, 32, 37, 38]. In contrast, the *"many-to-one" deployment model* executes all functions of a workflow within a shared sandbox [1, 4, 12, 30, 37]. Due to the recycling of runtime and efficient communication between parent and child processes, the *"many-to-one" model* can significantly reduce end-to-end latency compared to the *"one-to-one" model*.

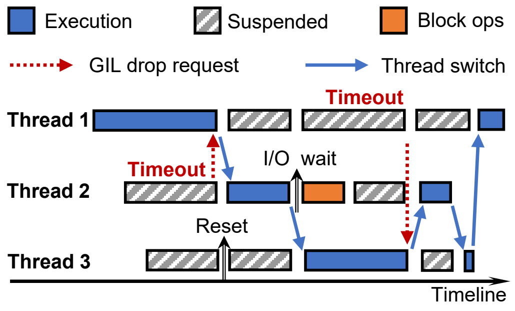

Figure 2: An example of GIL in Python.

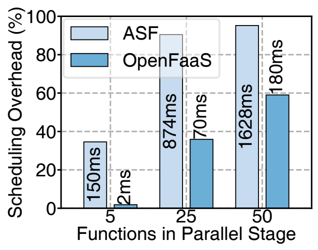

Figure 3: The scheduling overhead in FINRA.

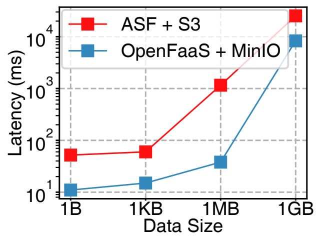

Figure 4: The transmission overhead.

However, even under the *"many-to-one" model*, the *pseudo parallelism* execution of code inherent in some interpreted languages makes it difficult for *parallel stages in workflows* to take full advantage of multi-core processors [14, 15, 30].

Pseudo-parallelism in Python: To ensure thread safety, CPython [16] employs a global interpreter lock (GIL) [3, 15] to prevent multiple threads from using CPUs concurrently. Figure 2 presents an example of how GIL is switched. Only the thread holding the GIL is permitted to execute its bytecodes, and it is asked to drop the GIL when others have been suspended for a specific timeout. Moreover, the thread actively drops the GIL during I/O operations to prevent potential long blocking. Therefore, multiple processes have to be used to execute parallel tasks due to the exclusive GIL in each process. Although there have been studies attempting to remove the GIL, they are either incompatible with CPython extensions [13, 17] or slow down the single-threaded performance [19, 24, 48].

Pseudo-parallelism in Node.js: Node.js executes JavaScript code in a single execution thread, which handles asynchronous I/O in parallel but suspends other tasks when CPU operations occurs, resulting in similar pseudo-parallelism problem as that in CPython. To remedy this, the worker threads [14] module was introduced to enable multiple threads in Node.js for parallel task execution. However, our evaluation in AWS Lambda reveals that worker threads incur more than 50 ms of startup overhead for each function, leading to doubled latency due to the median 60 ms latency of serverless functions [50].

2.2 Observations

In this section, we study the performance of existing "one-to-one" model and "many-to-one" model-based serverless workflow systems and find that the current deployment models are ill-suited for the serverless workflow.

We evaluate the performance of both commercial and open-source systems as below,

- *Amazon Step Functions (ASF)* [54]: The most popular orchestration service for serverless workflows in commercial clouds and adopts the *"one-to-one" model*.
- *OpenFaaS* [36]: A popular open-source platform that also adopts the *"one-to-one" model* (Table 2).
- *Faastlane* [30]: The state-of-the-art method of the *"many-to-one" model*, which uses processes for executing concurrent functions and threads for sequential functions.
- *Faastlane-T*: Using threads only for both concurrent and sequential execution. It is designed for evaluating the performance under thread-only configurations.
- *Faastlane*$^{+}$: Running 5 function processes fixedly in each sandbox, following the *"m-to-n" model*. It is designed for evaluating the performance under process-only configurations.

We use the *Financial Industry Regulatory Authority* (FINRA) application [2, 30] as the benchmark, with 5, 25 and 50 concurrent functions in the parallel pattern for validating trades for pre-determined rules. To ensure a fair comparison, we configure the respective Lambda functions such that their execution time matches those in our local cluster.

Observation 1: *The "one-to-one" model is characterized by significant scheduling and interaction overhead.*

We evaluate the performance of FINRA in *ASF* and *OpenFaaS*, and find the scheduling overhead can dominate the overall latency. As shown in Figure 3, *ASF* uses 150ms for scheduling a function, and it only able to run up-to 10 functions concurrently. Then, the scheduling overhead continues increasing, accounting up-to 95% of overall latency in 50 parallel functions. Although our local orchestrator in *OpenFaaS* experiences significantly less overhead for scheduling a function, the ratio for scheduling 50 functions can still achieve 59% of the end-to-end latency.

We further evaluate the latency of data exchange between functions across various sizes using third-party storage services (e.g., S3 [53] and MinIO [40]). Our results, shown in Figure 4, demonstrate that even the smallest data transfer can take up to 52ms due to multiple data copies and limited bandwidth in AWS Lambda [4, 46]. For 1GB data, the overhead can reach up-to 25s, significantly longer than the millisecond-scale functions. Despite the efficient network performance in local cluster, the interaction overhead still range from 10 ms to 10s, making it challenging to meet the latency SLO of latency-sensitive applications.

Observation 2: *The "many-to-one" model entails a significant block and startup overhead during function execution.*

The *"many-to-one" model* deploys multiple functions within one sandbox [1, 4, 30, 37, 56]. The functions can be implemented by either *processes* or *threads*. Although *threads* cloned from existing process are more efficient in terms of creation, context switching and communication due to resource share of process, they suffer the pseudo-parallelism problem in most serverless runtimes. In contrast, multiple *processes* fork individual execution context for

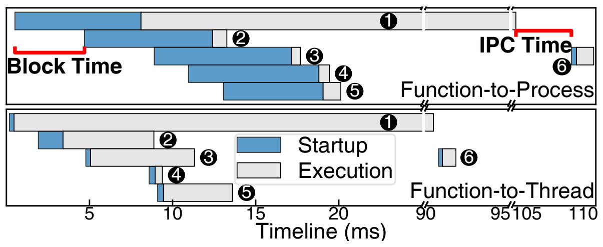

Figure 5: Timelines of Process and Thread execution mode for FINRA with 5 parallel functions.

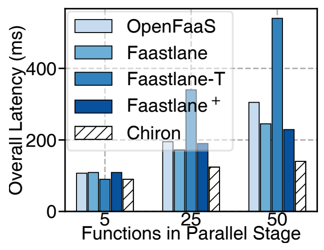

Figure 6: The comparison of end-to-end latency.

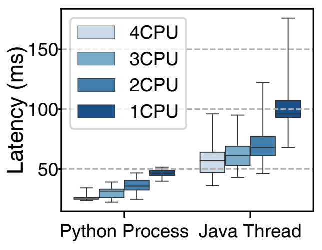

Figure 7: The distribution of latency w/o GIL.

each parallel task to achieve true parallelism, and dominate current *"many-to-one" model.*

We execute FINRA-5 under *Faastlane* to asses performance of processes based *"many-to-one" model*, and illustrate the execution timeline in Figure 5 (top). The interaction between functions through inter-process communication takes only 4.3ms, reducing latency by 71% compared to the *"one-to-one" model.* While the *startup time*, which refers to the time taken from forking current process to the beginning of function execution, is much shorter than the time required to spawn a sandbox, the average startup time (i.e., 7.5 ms) can be 10× higher than the execution time of sub-millisecond scale functions (e.g., ❷-❺). Moreover, subsequent processes need to wait for the completion of forking all previous processes. Therefore, the *block time* (e.g., ❸-❺), which refers to the waiting time involved when forking multiple processes sequentially, is 1×-2.1× longer than the startup time. For example, when 50 parallel functions execute simultaneously, the blocking time can reach up to 169 ms, similar to cold start overhead [63]. Thus, the *"many-to-one" model* built upon multiple processes is still far from optimal performance efficiency.

Observation 3: *The combined processes and threads can further enhance the performance efficiency of the "m-to-n" model.*

To explore the optimal deployment model, we evaluate various methods that handle parallel patterns with different deployment models and execution modes (i.e., process or thread). And the overall latency is shown in Figure 6. Despite attempting to amortize block and startup overhead over multiple sandboxes, the performance improvement of *Faastlane*$^+$ over *Faastlane* is only 6.5% for FINRA-50. This can be attributed to the continued heavy overhead of multiple processes within a sandbox, as well as the additional interaction overhead between sandboxes.

Although the pseudo-parallelism in multiple threads even leads to 77% slower than *OpenFaaS* when executing 50 parallel functions, we observe that *Faastlane-T* outperforms others by 17.4% in FINRA-5. We further show the execution timeline of *Faastlane-T* under FINRA-5 in Figure 5 (bottom), and find the profits of little startup time outweigh the additional execution latency due to pseudo-parallelism. Besides, block operations (e.g., sleep, IO) can also execute concurrently with the thread holding GIL as shown in Figure 2. Thus, neither true nor pseudo-parallelism can prevail under all scenarios, motivating a combined approach. To this end, *Chiron*, which employs the *"m-to-n" model* combining processes and threads, achieves optimal performance in all cases, leading to 15.9%-74.1% reduction in latency compared to others.

Observation 4: *The uniform allocation mechanism in existing deployment model can lead to poor resource efficiency.*

We further measure the overall memory and CPU consumption of FINRA under various models, and present the findings in Figure 8. It is observed *OpenFaaS*, i.e., the *"one-to-one" model* utilizes the most resources due to uniform allocation for every function in the workflow. *Faastlane* employs the *"many-to-one" model* and recycles resources between functions, resulting in an average reduction of 85.5% and 7.5% in memory and CPU cost, respectively, compared to *OpenFaaS.* The significant reduction in memory is attributed to the severe memory redundancy between sandboxes for language runtime and libraries [41], e.g., 77.2% in FINRA. However, the *"many-to-one" model* allocates separate CPU resource for each parallel function to achieve true parallelism, limiting the improvement of resource efficiency. *Chiron*, employing the non-uniform resource allocation for combined true and pseudo-parallelism, further reduces 82.7% CPU and 8.3% memory cost compared to *Faastlane.*

There exist *"many-to-one" model* that built on true parallelism. For example, FAASM [56] and Photons [11]. In this case, deploying all functions with *threads* can achieve the optimal performance. However, uniform resource provisioning for parallel functions also lead to sub-optimal resource efficiency. We evaluate the performance of parallel functions under Python's *ProcessPoolExecutor* [18] and Java threads, which both support true parallelism. *ProcessPoolExecutor* uses a process pool to avoid significant startup overhead of forking new processes for function execution. Four parallel functions with various execution behaviors but similar latency are selected from SLApp [33] for evaluation, including *factorial*, *fibonacci*, *disk-io* and *network-io*. We measure the latency distribution with various numbers of CPUs and show the results in Figure 7. The results indicate that the latency with fewer CPUs (e.g., 3), namely, the combined true and pseudo-parallelism only results in an average increases of 11.7% (or 4.2 ms) in latency compared to the latency with 4 CPUs, i.e., the uniform resource allocation.

2.3 Implications

Our motivational example demonstrates that even in the absence of cold starts, the *"one-to-one" model* still suffers from poor performance efficiency due to the scheduling and interaction overhead (**Observation 1**). While the *"many-to-one" model* reduces these overheads through deploying all functions within a sandbox, it also introduces significant block and startup overheads for functions especially developed using interpreted languages (**Observation 2**).

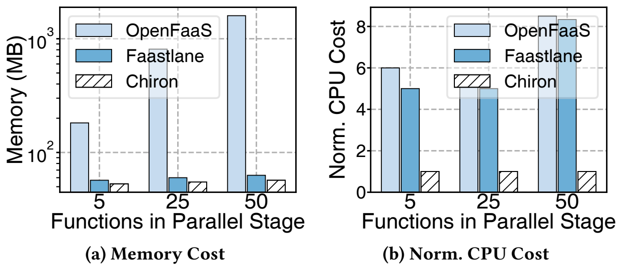

Figure 8: The comparison of overall resource consumption in FINRA.

As interpreted languages dominate current serverless applications (e.g., 58% are developed using Python and 31% are developed using Node.js [49, 50]), it becomes particularly critical and necessary to rethink the deployment model for serverless functions. Hence, we argue the *"m-to-n" model*, which can enhance both performance (**Observation 3**) and resource efficiency (**Observation 4**) through function partition and the combined usage of processes and threads.

3 DESIGN

In this section, we present the design of *Chiron*, which explores the optimal *wrap* to implement the *"m-to-n" model* for serverless workflow.

3.1 Overview

The basic idea of the *Chiron* is that *both resource efficiency and overall performance of serverless workflow can be optimized through the scheduling of the function partitions (i.e., wrap) and hybrid process-thread execution mode. Wrap* amortizes the possible excessive startup overhead of parallel processes over multiple sandboxes. Moreover, it mitigates the startup bottleneck of process by introducing thread execution, thereby achieving a balance between startup overhead and execution performance.

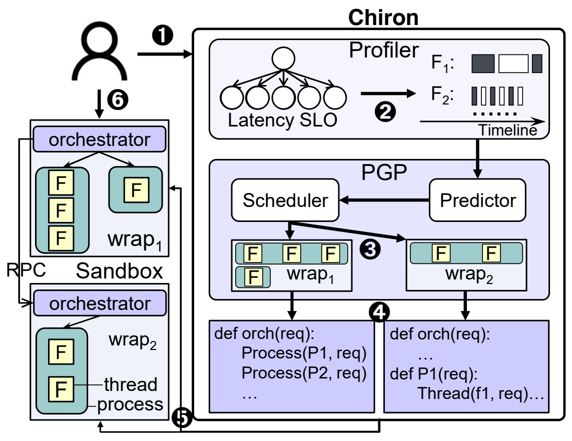

Figure 9: The design overview of *Chiron*.

Figure 9 illustrates the design overview of *Chiron*, which leverages the unique characteristics of workflows, such as startup overhead, interaction overhead and functions execution behavior, to achieve the optimal *wrap* design. This includes determining which functions should be grouped together in each *wrap* and the execution mode for each function within the *wrap*. Specifically, after the submission of the workflow definition (e.g., DAG, state machine) and latency requirement ❶ by the user, the *Profiler* module collects the execution behavior of each function, and extracts CPU and IO periods during function execution to facilitate performance prediction ❷. Then, the *PGP* algorithm utilizes the *Predictor* to explore the optimal *wrap* design ❸, maximizing resource efficiency while guaranteeing the end-to-end latency SLO. *Chiron* subsequently generates the orchestrator code for each *wrap* to manage function execution and state transfer ❹. The orchestrator is bundled with the *wrap*'s functions and deployed as a "new function", and the serverless platform spawns a separate sandbox for each *wrap*❺. Finally, the workflow invocation is directed to the first *wrap* which invokes subsequent *wraps* to collaboratively handle the request ❻.

3.2 Profiler

Motivated by the GIL switching mechanism shown in Figure 2, the switching process can be simulated based on the CPU and block periods (i.e., duration for IO operations) in function execution. Specially, *Chiron* utilizes the *strace* syscall to extract the block periods from block syscalls invoked during function execution, such as *open*, *read*, *write*, *poll*, *select*, *sendto*, and others. And the others can be considered CPU time.

Figure 10 shows an example of function execution behaviors extraction utilizing *strace*. The *handle* function calls several syscalls, including *select*, *write* and *read*, and *strace* records the start timestamp, name and duration of each syscall. We can then deduce CPU and block periods based on the function execution start timestamp. To mitigate the estimation error introduced by the profiling overhead of *strace*, *Profiler* scales down all block periods based on the average function latency recorded without *strace*.

```
# function start timestamp: 0 ms
def handle(req):                # strace log
    ...                         # syscall start timestamp, syscall, duration
    time.sleep(1)               # 48 ms, select(), 1001 ms
    path = "test.txt"
    ...
    with open(path, "w") as f:
        f.write("1")            # 1070 ms, write(</home/app/test.txt>), 0.042 ms
    ...
    with open(path, "r") as f:
        f.read()                # 1081 ms, read(</home/app/test.txt>), 0.025 ms
    ...
# block period 1: (48 ms, 48 + 1001 ms)
# block period 2: (1070 ms, 1070 + 0.042 ms)
# block period 3: (1081 ms, 1081 + 0.025 ms)
```

Figure 10: Example of extracting block periods using *strace*

In summary, *Profiler* collects the function latency, start and end timestamp of each block syscall in solo-run case for each function. These data are then utilized for predicting the overall latency of multiple threads within a process as shown in Algorithm 1.

3.3 Predictor

Designing an adaptive and dynamic performance predictor using traditional machine learning (ML) or deep learning (DL) algorithms requires considerable expert knowledge in concepts such as feature engineering and graph embedding of various workflows. Additionally, a substantial number of training samples are typically required to achieve high accuracy, which leads to severe profiling costs. To circumvent these issues, we utilize a white box method.

Serverless workflows comprise a sequence of execution stages, wherein each stage includes one or more parallel functions. The end-to-end latency of workflow can be estimated by summing the execution time of all stages, as follows,

$$T_{workflow} = \sum_{i=1}^{n} T^{i}_{stage} \tag{1}$$

The execution time of stage i is decided by the slowest *wrap*, while other *wraps* are invoked by *wrap1* through the network communication.

$$T^{i}_{stage} = \max\left(T^{i,1}_{wrap}, \max_{k>1}(T^{i,k}_{wrap} + (k-1)\times T_{INV}) + T_{RPC}\right) \tag{2}$$

where $T^{i,k}_{wrap}$ denotes the latency of the kth *wrap* in stage i, T_{INV} denotes the overhead of multiple invocation resulting from platform and programming library, and T_{RPC} denotes the network communication overhead for one invocation.

The latency of each *wrap* consists of both computation time and interaction time between processes,

$$T^{i,k}_{wrap} = \max_{j\in P^{i,k}_{wrap}} T^{i,k,j}_{P} + T_{IPC}\times(|P^{i,k}_{wrap}| - 1) \tag{3}$$

where $T^{i,k,j}_{P}$ denotes the latency of the jth process in $wrap_{i,k}$, T_{IPC} denotes the communication overhead with another process through Linux pipe, and $P^{i,k}_{wrap}$ denotes the set of processes in $wrap_{i,k}$. We suppose there is no interaction time for thread communication within a process due to the shared memory.

According to **Observation 2**, the overall latency of a process can be estimated as follows,

$$T^{i,k,j}_{P} = (j-1)\times T_{Block} + T_{Startup} + T^{i,k,j}_{exec} \tag{4}$$

where T_{Block} and $T_{Startup}$ indicate the block time and startup overhead for forking a new process respectively, and are both estimated with constant values. $T^{i,k,j}_{exec}$, the overall latency of executing multiple threads within a process, can be estimated through simulating GIL switching based on the mechanism shown in Figure 2 and function behaviors extracted by *Profiler*.

Next, we show how the execution latency of a process in a *wrap*. namely, the overall latency of multiple threads within the process, is predicted in Algorithm 1. The main thread is responsible for creating threads, and it is assumed that the same amount of functions is started in each interval (Lines 4-5). When turning to a function, the thread executes continuously until timeout (Lines 8-9) or a block operation occurs (Lines 14-16). When a function completes, the corresponding thread is destroyed and removed from the scheduling list (Lines 10-11). To emulate the scheduling of the operating system, the thread with the minimum CPU time from the non-block threads is selected to hold the GIL according to the *Completely Fair Scheduler* [43] (Line 17) at the end of each interval. Finally, we derive the total latency of multiple threads when all of them have finished (Lines 12-13).

Algorithm 1: Multi-Threads Latency Prediction

```
Data:
  $I$ ▷ The interval of switching threads;
Input:
  $fs$ ▷ functions $f_1, f_2, ..., f_q, ...$;
Output:
  $T_{exec}$ ▷ The execution latency of multiple threads within a process;
1  $T_{exec} \leftarrow 0$;
2  $f \leftarrow$ main thread; // the thread that holds GIL
3  while True do
4      if turn to main thread then
5          $f_x, f_{x+1}, ..f_{x+y} \leftarrow$ start $y$ functions in $I$;
6      else
7          // turn to function $f$
8          if no block op in this interval then
9              $T_{exec} \leftarrow T_{exec} + I$;
10             if $f$ is over then
11                 remove $f$ from the scheduling list;
12             if all threads are over then
13                 return $T_{exec}$;
14         else
15             $T_{avl} \leftarrow$ time before block;
16             $T_{exec} \leftarrow T_{exec} + T_{avl}$;
17     $f \leftarrow$ select the function which has the minimum CPU time
         and is not block;
```

3.4 Scheduler

Given the workflow profile and prediction model, we design *PGP* to determine the optimal number of *wraps*, as well as the processes and threads in each *wrap*. However, unlike the k-balanced graph partition problem in conventional distributed computing field, e.g., MapReduce, NFV, *wrap* partition for serverless workflows must address three new challenges: two-layer (i.e., sandbox and process) partition, unspecified amount of subsets, and dynamic cost of node and edge cuts. Therefore, existing algorithms can't be used directly for our problem. To solve above challenge, we propose a heuristic method based on the Kernighan-Lin partitioning algorithm [28].

The fundamental concept of *PGP* is to partition processes and threads iteratively guided by latency prediction. After an initial partition, predictions are continuously made and the *wrap* is repartitioned until the algorithm converges. Algorithm 2 shows the details. We utilize an incremental iterative method to find the minimum number of processes in *wraps* that can satisfy the given SLO (Line 1-5). In each iteration, *PGP* first determines the original *wraps* which minimize block overhead and network communication through maximizing the number of processes in *wrap1*, while only a single process in each of others (Lines 6-9). Subsequently, we employ Kernighan-Lin algorithm [28] to derive the optimal *wraps*, i.e., function partition with the lowest predicted latency (Lines 10-11). Finally, *PGP* heuristically divides processes into as few *wraps* as possible to achieve the optimal resource efficiency while ensuring the specified SLO (Lines 13-16).

The Kernighan-Lin algorithm attempts to find a series of optimal *element swapping* operations between each pair of *sets* to maximize cumulative gains. In *PGP*, a *set* refers to the collection of functions contained within a process, while *element swapping* refers to the swapping of functions between two processes. We iteratively selects functions that can minimize the predicted latency after swapping them between two function sets, and records the latency benefit of each swap operation (Lines 19-23). When all functions have been interchanged, we choose the first k swaps that can achieve the lowest predicted latency cumulatively, and apply those operations to the original function sets (Lines 24-25).

Algorithm 2: Wrap Scheduling

```
Input:
  G ▹ The stages of workflow;
  SLO ▹ The given SLO of workflow;
Output:
  P ▹ The function partitions;
  N ▹ The number of processes in wraps;
M ← the max parallelism of workflow;
P ← {}, N ← {};
for n in {1, 2, ..., M} do
    $T_{workflow} \leftarrow 0$;
    for stage i in G do
        // init the number of processes in each wrap
        $N_i \leftarrow \{min(\lfloor T_{RPC}/T_{Block} \rfloor, n), 1, ..., 1...\}$;
        // init partition of n processes
        $P_i \leftarrow \{\{f_1, f_{n+1}, ...\}, \{f_2, ...\}, ..., \{f_n, ...\}\}$;
        for function set $A \in P_i, B \in P_i$ do
            KernighanLin(A, B); // minimize latency
        $T_{workflow} \leftarrow T_{workflow} + T^i_{stage}$;
    if $T_{workflow} \leq SLO$ then
        for $stage_i$ in G do
            // find optimal $N_i$
            $N_i \leftarrow$ deploy max processes in each wrap;
        return P, N; // find solution
Function KernighanLin(A, B):
    $A', B' \leftarrow A, B$; // copy function set for swapping
    while $|A'| > 0$ and $|B'| > 0$ do
        $a, b \leftarrow$ swap $a \in A', b \in B'$ that minimize latency;
        $g \leftarrow T_{before} - T_{after}$; // record latency benefit
        $A' \leftarrow A' - a, B' \leftarrow B' - b$;
    $k \leftarrow \arg\max_k \sum_{i=1}^{k} g_i$; // choose k to minimize latency
    move $a_1, ..., a_k$ to B and $b_1, ..., b_k$ to A;
```

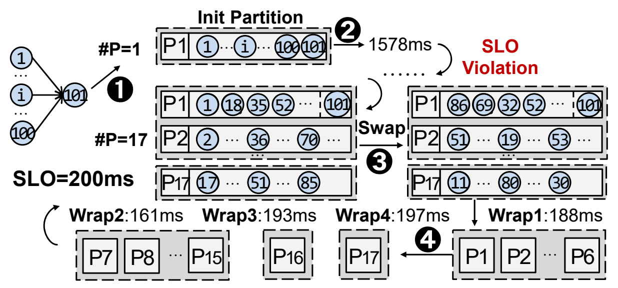

Figure 11: Scheduling FINRA-100 using *PGP*.

Figure 11 illustrates an example of *PGP* applied to the FINRA application with a degree of parallelism of 100. First, *PGP* attempts to execute all functions within a single process ❶, but the predicted latency of 1578 ms exceeds the SLO ❷. Subsequently, *PGP* explores the optimal partition by swapping functions between processes while increasing the number of processes ❸. Finally, *PGP* identifies the partition with the minimal latency of 197 ms using 17 processes, which meets the 200 ms SLO. Moreover, it divides the processes into 4 *wraps* ❹.

Notably, not all functions of a workflow can share the same sandbox due to conflict between language runtimes (e.g., Python 2 and Python 3) or package dependencies. Thus, some functions must still execute in separate sandboxes, namely, a *wrap* that only contains one function. Moreover, functions that need to process the same file cannot share sandbox, as it can lead to unexpected results. Figure 10 shows how *Profiler* traces files processed in during function execution, which can be used to measure if the functions need to be partitioned into different *wraps*.

Upon determining the optimal *wraps* using *PGP* offline, subsequent requests of the workflow can reuse these *wraps* to avoid the scheduling overhead. Additionally, the *Profiler* and *PGP* are re-run periodically to update *wraps*, enabling them to adapt to changes in the workload.

4 MEMORY ISOLATION BETWEEN THREADS

While the thread execution in *wrap* can improve both performance and resource efficiency, it raises privacy concerns as the unrestricted sharing among multiple threads can potentially lead to the exposure of sensitive data in trusted functions.

Software-fault isolation (SFI) [56, 62] provides memory isolation at the thread level by translating applications to WebAssembly and executes each function within a lightweight isolation abstraction, which has a dedicated thread and can access data in shared memory regions directly. On the other hand, *Intel Memory Protection Keys* (MPK) [25] can also achieve private arenas for each thread through assigning specific keys to pages and controls access rights to them, providing hardware support for page partitioning.

Isolation Mechanism	Startup Overhead	Interaction Overhead	Execution Overhead	
			Fibonacci	DiskIO
SFI	18 ms	8 ms	52.9%	29.4%
Intel MPK	0.2 ms	0	35.2%	7.3%

Table 1: The Comparison of SFI and Intel MPK.

We conducted an evaluation of SFI and Intel MPK using a Python application on our testbed (Table 2), and show the results in Table 1. Our findings indicate that SFI introduces much more startup and interaction overhead compared to process. With regards to function executions, both WebAssembly [26] and Intel MPK [45] incur more instructions and performance degradation, but Intel MPK suffers less overhead than WebAssembly. Therefore, in case of memory isolation is required, we choose Intel MPK to enhance memory privacy for thread execution in *Chiron*. For a fair comparison, *Chiron* only employs thread execution with Intel MPK for sequential functions and forks multiple processes to execute parallel functions in our evaluation.

True Parallelism. Although the combined process and thread can improve resource efficiency, it is still subject to additional execution latency arising from the GIL. To achieve the optimal performance, *Chiron* necessitates an execution mechanism that facilitates true parallelism for all functions.

There have been several attempts to remove the GIL from Python [19, 24, 57], but such efforts incur more than 10% overhead in single-threaded performance, in addition to compatibility issues [24]. Therefore, we continue to employ multiple processes to support true parallelism, but uses a process pool instead of forking new processes for each request to avoid the long startup overhead.

Due to the little startup overhead and true parallelism in a process pool, we can deploy all functions within a workflow in a single *wrap* (i.e., $n = 1$ for the *"m-to-n" model*) to avoid network cost. Within the pool-based *wrap*, *Chiron* enables CPU sharing between processes by setting affinity of each process to a specific CPU to derive the optimal resource efficiency.

5 IMPLEMENTATION

We implement *Chiron* in Python 3.11 atop OpenFaaS[36], a widely used open-source serverless computing platform. The components including *Profiler*, *Predictor*, *Scheduler* and *Generator* are all non-invasive plug-ins in the form of web services built upon *Flask* [44]. *Chiron* is publicly available at https://github.com/tjulym/Chiron.

Profiler: After receiving workflow submission from user, *Profiler* obtains the pid of each function's sandbox through APIs provides by Docker and *gateway* in OpenFaaS. Then, it employs the *subprocess* module of Python to invoke *strace* syscall to profile the latency and block periods of each function under solo-run.

Scheduler: It can use multiple processes to explore *wrap* partition under various number of processes in parallel to improve scheduling efficiency. It also adopts *cgroups* to allocate CPUs for *wraps*.

Generator: It produces the orchestrator code as the function entry (e.g., *handler.py* of the python template in OpenFaaS), uses the *psutil* [51] library to set CPU affinity for each process to isolate resources within a *wrap*. For a better performance efficiency, we use *python3-flask* template to deploy *wraps*, which employs *of-watchdog* [35] that supports HTTP proxying instead of *classic-watchdog* [34] that forks a new process for each request. To support Intel MPK, we also create a new template in OpenFaaS that contains the mpk-memalloc-module [30], and build images with *wrap* codes. As for process pool, *wrap* starts *ProcessPoolExecutor* when initializing sandbox based on the official Python library [18].

6 EVALUATION

	Configuration
Hardware	CPU: Intel Xeon Gold 6230 @2.1GHz * 40 DRAM: 128GB, Disk: 960GB SSD, Nodes: 8
Software	Operating system: Ubuntu 16.04 Docker server and client version: 20.10.7 Kubernetes server and client version: 1.23.6

Table 2: Experimental testbed configuration.

Testbed and Benchmarks: We conduct the experiments on a local cluster with configurations shown in Table 2. The benchmarks employed are summarized as follows:

- Social Network [23] (SN) comprises 4 stages and 10 functions, with the maximum parallelism of 5.
- Movie Reviewing [23] (MR) comprises 4 stages and 9 functions, with the maximum parallelism of 4.
- SLApp is produced from [33] and comprises 2 stages and 7 functions, each has similar latency but falls into three workload types, i.e., CPU intensive, disk I/O intensive, and network I/O intensive. Note that there is no sequential function in SLApp, with the maximum parallelism set at 4.
- SLApp-V is a variant of SLApp, which is also generated from [33] and comprises 5 stages and 10 functions, with the maximum parallelism of 5.
- Financial Industry Regulation (FINRA) [2, 30] comprises 2 stages, and we configure varying numbers of parallel functions, including 5, 50, 100 and 200, for evaluation.

Metrics and comparison algorithms: Our evaluations center around the following metrics: (1) the accuracy of performance prediction, (2) end-to-end workflow latency, and (3) resource efficiency and dollar cost. we evaluate *Chiron* against several popular commercial and open-source serverless platforms. This includes *AWS Step Functions* [54] and *OpenFaaS* [36], which both adopt the *"one-to-one" model*, as well as *SAND* [1] and *Faastlane* [30], which are based on the *"many-to-one" model.* In particular, *SAND* executes each function in a separate process, while *Faastlane* uses thread execution in a sequential function to reduce the function interaction overhead. We also evaluate the performance of *Chiron* and *Faastlane* using Intel MPK (*Chiron-M* and *Faastlane-M*) and process pool (*Chiron-P* and *Faastlane-P*). Notably, process pool can achieve similar performance to that of multiple threads without the GIL. Throughout our evaluation, we use a whole CPU as the allocation unit, and for a fair comparison, we configure respective functions in *AWS Lambda* such that the execution time of *ASF* matches that of our local cluster.

6.1 Prediction Error

The prediction error is defined as $(\hat{P} - P)/P$, where $\hat{P}$ and P denote the predicted latency and the actual value, respectively. We compare the *Predictor* of *Chiron* against *Random Forest Regression (RFR)*, *Long Short-Term Memory (LSTM)*, and *Graph Neural Network (GNN)*. We exploit all possible *wraps* in SN, MR, SLApp, SLApp-V and FINRA-5 applications using native thread, Intel MPK and process pool implementations for evaluation. And the details of models are summarized as follows:

- *RFR* is employed to predict the latency of multiple threads as Algorithm 1, and the features utilized include each function's solo-run latency, *Context-switches, L1I MPKI, LID MKPI, L2 MPKI, L3MPKI, TLBD MPKI, TLBI MPKI, Branch MPKI, MLP, CPU utilization, Memory utilization, Network bandwidth, LLC, IPC, Disk IO and Memory IO*, which are recommended by Gsight [65]. And this model is built using *RandomForestRegressor* [52] module from *scikit-learn* library, with default parameters.
- *LSTM* shares the same input and output as *RFR.* It is constructed based on the *LSTM* [21] module from *PyTorch.* We evaluate the *LSTM* model with multiple learning rates, including 0.1, 0.01, 0.001, and 0.0001, and find that 0.01 yields the best accuracy. Therefore, we set the learning rate to 0.01 for our experiments. On the other hand, we set the batch size to 1, while keeping the other hyper-parameters at their default values.
- *GNN*'s input consists of a feature matrix that utilizes the aforementioned function features, as well as an adjacency matrix that represents the relationships between threads, processes, stages, and workflows within *wrap.* And it outputs the end-to-end latency of workflow. We use *PyTorch* to implement this model with hyper-parameters recommended by *Eagle* [10].

Prediction accuracy: *Chiron* averages 6.7% error in predicting the overall latency. Figure 12 shows the distribution and average value of the prediction errors across various models. The mean absolute prediction errors of *Chiron* range from 1.4% to 14.2%. In most cases, our predictor exhibits the highest precision, reducing the prediction error by 78.1%, 86.6% and 70.1% on average, when compared to *RFR*, *LSTM* and *GNN*, respectively. While Learning

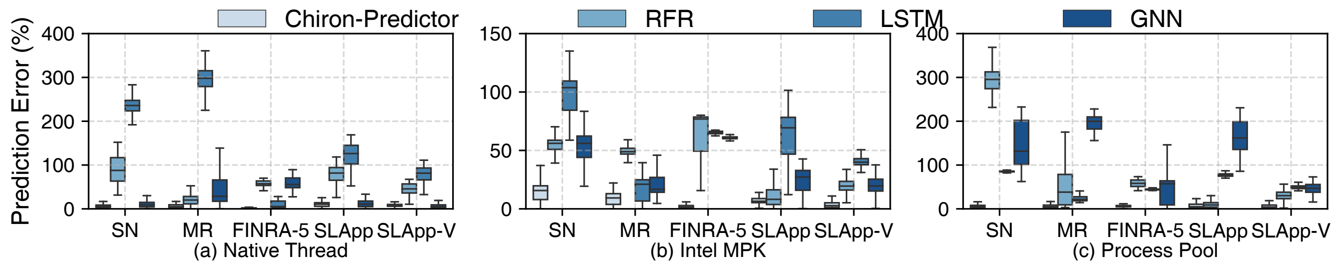

Figure 12: The prediction error of serverless workflows under various prediction models.

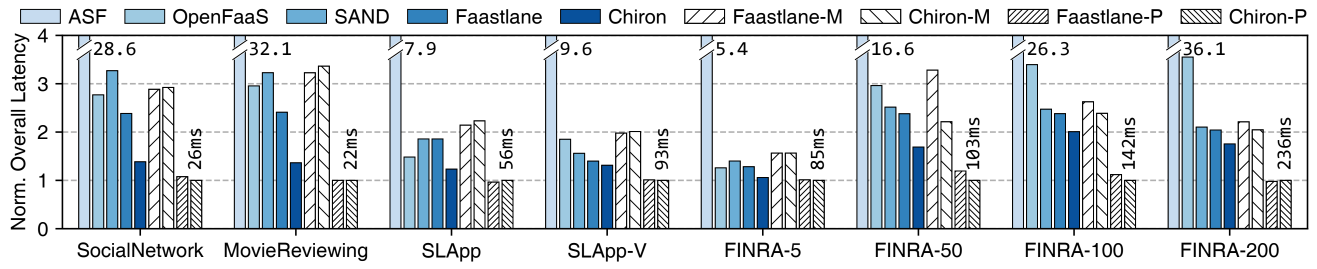

Figure 13: The comparison of normalized workflow end-to-end latency under various deployment models.

models sometimes achieve better precision than our *Predictor* (e.g., *GNN* for SLApp-V), their lack of diversity in training data, including various structures of workflows and function workloads, can limit their applicability. Overall, our *Predictor* achieves 6.7% prediction error on average with little profiling overhead, making it much more efficient than building learning models.

6.2 Overall Performance

End-to-end workflow latency: *Chiron* can respectively reduce the overall latency of workflow by up-to 53.8% and 43.4% over *OpenFaaS* and *Faastlane*. We measure end-to-end latency by executing each workflow without cold start at least 10 times and show the average values in Figure 13. The SLO for *Chiron* is defined as the average latency of *Faastlane* with an additional 10 ms slack. Our results show that the scheduling latency of *ASF* for FINRA-200 reaches more than 8s, highlighting the need to increase the deployment granularity for latency-sensitive applications. While *OpenFaaS* demonstrates better performance than *SAND* in some cases due to efficient network communication in local cluster, the scheduling and interaction overhead in the *"one-to-one" model* continues to increase with the increment of parallel functions. For MPK and pool-based methods, wherein all parallel functions have the same execution mode, *Chiron* can still achieve similar and even better performance. Overall, *Chiron* reduces the end-to-end latency by an average of 89.9%, 37.5%, 32.1% and 25.1% over *ASF*, *OpenFaaS*, *SAND* and *Faastlane* with native thread, respectively.

SLO violation: *Chiron* can guarantee the latency SLO of workflows. Figure 14 shows the SLO violation rate of *Chiron* averages at 1.3%, which is significantly lower than that of *Faastlane*. In fact, *Chiron* adopts larger parameters to estimate the latency, avoiding performance violation resulting from mispredictions.

Startup overhead: *Chiron* can significantly reduce startup overhead for parallel functions while introducing little effect to function execution. Figure 15 employs the latency CDFs of the

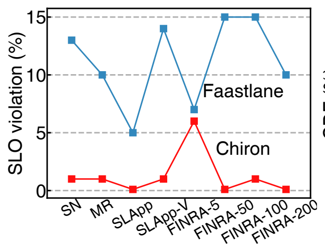

Figure 14: The comparison of SLO violation rate.

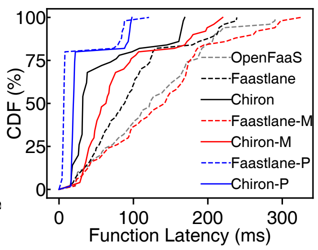

Figure 15: The latency CDF of FINRA-50.

parallel functions to show the influences of different deployment models on startup overhead and function execution. The blue dotted line (*Faastlane-P*) has the lowest startup time due to the usage of process pool. However, when there are an excessive number of functions in the process pool, long-running functions may suffer block and scheduling overhead, leading to long-tail execution in the last phase. On the other hand, all *Chiron* (solid lines) methods start functions and complete execution faster than others. While *Chiron-P* introduces more startup overhead for some functions due to the share of CPUs, long-running functions are started preferentially to mitigate the influence of skew execution and achieve the optimal latency among all systems. Overall, *Chiron* is up to 32.5% faster than *Faastlane-M* and *Faastlane-P* regardless of whether suffering from startup overhead.

6.3 Resource Efficiency

Memory consumption: *Chiron* can respectively save up-to 97% and 22% memory resources over *OpenFaaS* and *Faastlane*. Figure 16 shows the memory consumed under different platforms. *OpenFaaS* always requires the maximum memory due to the redundancy of language runtime and libraries between sandboxes in the *"one-to-one" deployment model.* When compared with

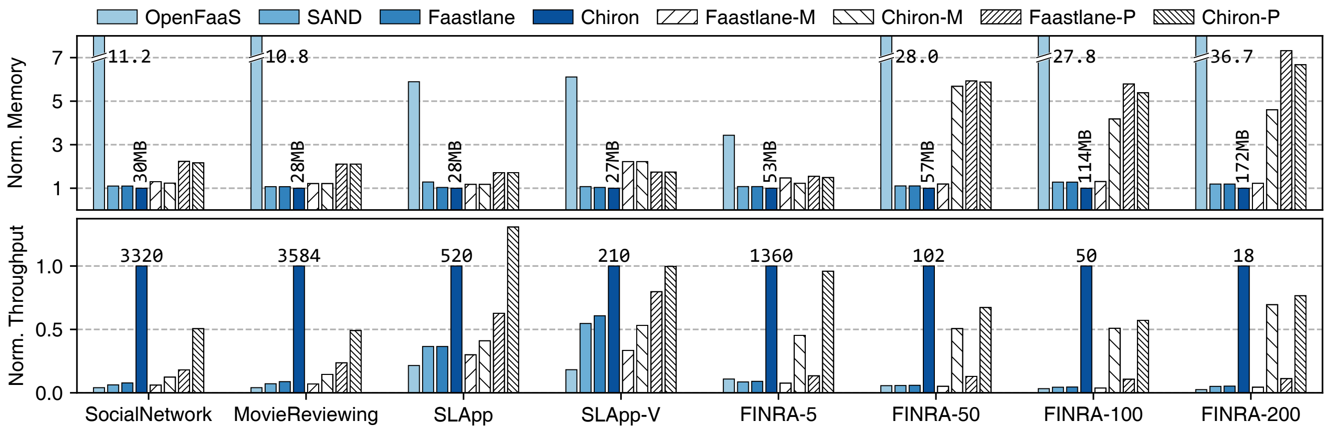

Figure 16: The comparison of normalized memory consumption and maximum throughput (req/s) in a worker node. *Chiron* can improve system throughput by 1.3×-39.6×.

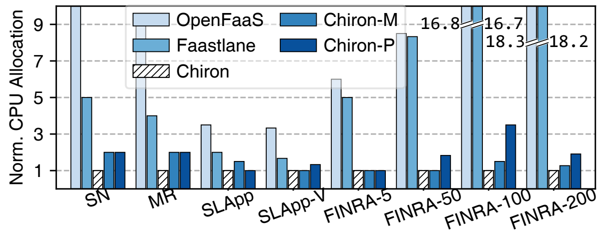

Figure 17: The comparison of allocated CPUs. *Chiron* can save to 20%-94% CPU resources.

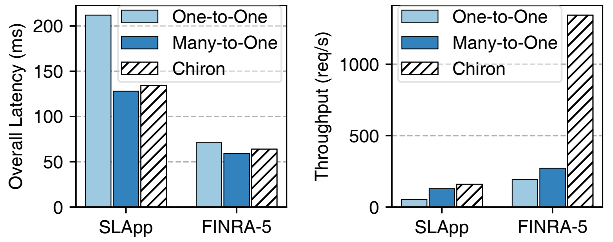

Figure 18: The comparison in Java. *Chiron* can improve system throughput by up-to 4.9× under true parallelism.

more processes in the *"many-to-one" model*, the shared memory between multiple threads help *Chiron* to further reduce 10% memory consumption on average compared with *Faastlane*. *Chiron-M* amortizes startup overhead in 6, 9 and 14 sandboxes for FINRA-50, FINRA-100 and FINRA-200 respectively, leading to 2.2×-3.7× memory cost than *Faastlane-M*. For systems using process pool, the long-running processes consumes more than 5× memory to avoid duplicate startup overhead.

CPU consumption: *Chiron* can save up-to 94% CPU resources over *OpenFaaS* and *Faastlane*. Figure 17 reports the normalized number of allocated CPUs under various platforms. *Faastlane* recycles CPUs between stages instead of allocating separate CPU resource for each function in *OpenFaaS*. But it requires the same number of CPUs as the maximum parallelism in DAG to ensure true parallelism. In comparison, *Chiron* explores the minimum number of CPUs while guaranteeing latency SLO through function partition which makes full use of trade-offs between true and pseudo-parallelism. As we can see, *Chiron* can reduce 75%, 66% and 63% CPU cost than *Faastlane* with native thread, Intel MPK and process pool, respectively.

Throughput: *Chiron* can improve system throughput by up-to 39×. Given the limited resources in each worker, we show the normalized maximum RPS in Figure 16. Due to the both better performance and resource efficiency, *Chiron* can improve the throughput by 12.2×, 6.5× and 4.1× on average compared with *Faastlane*, *Faastlane-M* and *Faastlane-P*, respectively. Although pool-based systems experience the optimal latency because of native true parallelism, the higher resource requirements results in lower throughput than *Chiron* with native threads in most cases.

No GIL: *Chiron* can also improve throughput by up-to 4.9× w/o GIL. We further evaluate the performance of serverless workflows in Java, enabling the true parallelism of native threads, and show the results in Figure 18. Although *Chiron* is reduced to only thread execution, it can also achieve 5× and 3.1× system throughput over the *"one-to-one" model* and *"many-to-one" model*, respectively, due to the better resource efficiency.

Cost efficiency: the *"m-to-n" deployment model* can help developers reduce the cost of workflow by 23.1%-99.6%. We derive the dollar cost of each methods based on $0.0000025 for GB-Second memory and $0.0000100 for GHz-Second CPU [7], and show the normalized costs in Figure 19. The *"one-to-one" model* has to additionally pay for every state transition between functions [54], then costs 272× over *Chiron* at most. Due to performance and resource advantages, *Chiron* reduces 44.4%-95.3% dollar cost than *Faastlane*. Although *Chiron-M* and *Chiron-P* consume much more memory, the significant reduction in CPU allocation still helps to save 65.2% and 62.1% cost on average compared to *Faastlane-M* and *Faastlane-P*, respectively. *Chiron* can lead to higher cost reduction when CPU scales proportionally with memory due to the over-provisioning of memory. In short, the *"m-to-n" model* is the most cost efficiency among all deployment models.

Resource overhead: *Chiron* generates negligible overhead in system resource consumption. Each component in *Chiron* consumes no more than 40MB memory. For CPU cost, both *Profiler* and *Generator* use less than 0.1 core. *Chiron* allocate 1 core for the *PGP* module by default, which can also use more cores to explore multiple partitions in parallel to improve the scheduling efficiency.

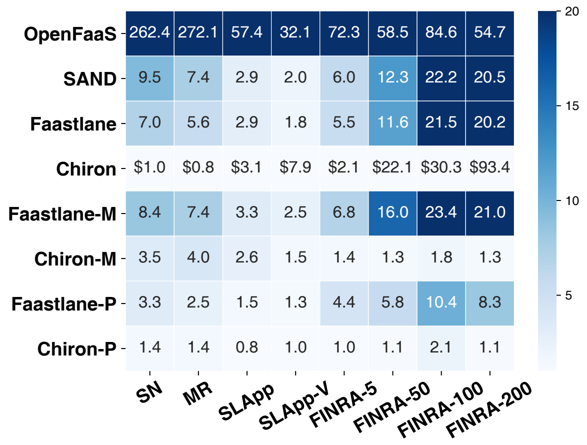

Figure 19: The comparison of cost (in USD) per 1 million workflow requests normalized by *Chiron*. *Chiron* can save dollar cost by up-to 99.6%**.**

7 DISCUSSION

Scalability. Although *PGP* can incur minute-level overhead when orchestrating large workflows (e.g., hundreds of functions), *wrap* scheduling operates in an offline manner and does not affect the real-time response latency of requests. Specifically, the scalability of *PGP* may be limited by three factors: (1) Iterative exploration: exploring the desired number of processes incrementally. This module can be parallelized to speed up scheduling in large workflow scenarios. (2) Kernighan-Lin algorithm: exploring the optimal partition given a specific number of processes. This module exhibits the highest complexity, but still operates in an offline way. Furthermore, other optimization algorithms can also be employed for partitioning. (3) Prediction: estimating the latency of multiple threads. This component maintains sub-millisecond overhead even in scenarios with hundreds of threads and exhibits good scalability. Additionally, when there are an excessive number of *wraps* in a workflow, the current centralized scheduling architecture of *Chiron* can lead to high real-time request scheduling overhead, similar to the *"one-to-one"* model. Decentralized scheduling, which offloads function scheduling to each worker node, is orthogonal to *Chiron* and can help mitigate this issue.

Application scenario. Although *Chiron* has demonstrated remarkable improvements in parallel-structured DAG workflows, such as average latency reduction of 25% and 12× increase in throughput compared to existing methods, further investigation and research are still warranted in the following scenarios. (1) Incompatible functions: where runtime or dependency conflicts prevent functions from executing concurrently within the same instance. (2) Dynamic DAGs: where the function chain of workflow is not known a priori, such as *switch* step in Video-FFmpeg [6] determines whether to execute the *split* function or the *simple_process* function based on the result of the *upload* function.

8 RELATED WORK

One-to-One model. The current serverless platforms deploy functions in separate sandboxes which leads to frequent cold start and high interaction latency in workflow. Pocket [29], Crucial [46], and Cloudburst [58] adopt specialized distributed remote stores to mitigate interaction overhead. Nightcore [27] executes functions on the same server and designs message channels for IPC to reduce RPC overhead. FaaSFlow [32] partitions the workflows into subgraphs based intermediate data size, and enables the direct data movement through in-memory storage. Xanadu [8] and ORION [38] pre-warm functions to avoid cascading cold starts in workflow. Although these systems do obtain performance improvement, the interaction overheads and resource redundancy are still significant.

Many-to-One model. Some researchers enlarge the deployment granularity through executing functions within a workflow in a shared sandbox. SAND [1] designs application-level sandboxing and executes functions in separate process to provide lower latency. WuKong [4] adopts decentralized scheduling to reuse executors to enhance the data locality. SONIC [37] transparently selects the optimal data-passing method for each edge of a serverless workflow DAG. Faastlane [30] uses thread execution where function can communicate through memory load/store instructions to minimize function interaction latency even further. However, we observe that these methods have not taken into account the block latency, where subsequent processes need to wait for the completion of forking all previous processes. And this overhead can even be comparable to cold start, leading to sub-optimal performance. Although some runtimes [11, 27, 56] allow true parallelism of multiple threads, the uniform resource allocation mechanism overlooks the workload heterogeneity, and leads to severe resource inefficiency.

9 CONCLUSION & FUTURE WORK

This paper presents *Chiron*, a new *"m-to-n" deployment system* for efficient serverless workflow by partitioning functions into multiple *wraps*, and executing them with combined processes and threads. *Chiron* has two key attributes. (1) High performance: *Chiron* can reduce startup overhead significantly while introducing little influence on interaction and function execution. (2) Resource efficiency: *Chiron* exploits the deployment granularity of *wrap* with the minimum amount of CPUs while guaranteeing the latency SLO.

In the future, we would like to expand *Chiron*'s capabilities to public cloud scenarios by incorporating strong yet lightweight isolation mechanisms.

10 ACKNOWLEDGMENTS

Our sincere thanks and appreciation goes to the SC reviewers and our shepherd, for their extremely insightful comments and suggestions that have significantly improved the quality of this paper. This work is supported by the National Key Research and Development Program of China No.2022YFB4500702; project ZR2022LZH018 supported by Shandong Provincial Natural Science Foundation; the National Natural Science Foundation of China under grant 62141218, 62372322 and the open project of Zhejiang Lab (2021DA0AM01/003).

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 2018, Boston, MA, USA, July 11-13, 2018*, Haryadi S. Gunawi and Benjamin Reed (Eds.). USENIX Association, 923–935. https://www.usenix.org/conference/atc18/presentation/akkus

[2] United States Financial Industry Regulatory Authority. 2023. Finra Case Study. https://aws.amazon.com/cn/solutions/case-studies/finra-data-validation/.

[3] David Beazley. 2009. Inside the Python GIL. http://www.dabeaz.com/python/GIL.pdf.

[4] Benjamin Carver, Jingyuan Zhang, Ao Wang, Ali Anwar, Panruo Wu, and Yue Cheng. 2020. Wukong: a scalable and locality-enhanced framework for serverless parallel computing. In *SoCC '20: ACM Symposium on Cloud Computing, Virtual Event, USA, October 19-21, 2020*, Rodrigo Fonseca, Christina Delimitrou, and Beng Chin Ooi (Eds.). ACM, 1–15. https://doi.org/10.1145/3419111.3421286

[5] Alibaba Cloud. 2023. Serverless Workflow: Visualization, O&M-free orchestration, and Coordination of Stateful Application Scenarios. https://www.alibabacloud.com/product/serverless-workflow.

[6] Alibaba Cloud. 2023. Use FFmpeg in Function Compute to process audio and video files. https://www.alibabacloud.com/help/en/function-compute/latest/use-ffmpeg-in-function-compute-to-process-audio-and-video-files.

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

[8] Nilanjan Daw, Umesh Bellur, and Purushottam Kulkarni. 2020. Xanadu: Mitigating cascading cold starts in serverless function chain deployments. In *Middleware '20: 21st International Middleware Conference, Delft, The Netherlands, December 7-11, 2020*, Dilma Da Silva and Rüdiger Kapitza (Eds.). ACM, 356–370. https://doi.org/10.1145/3423211.3425690

[9] 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 *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, James R. Larus, Luis Ceze, and Karin Strauss (Eds.). ACM, 467–481. https://doi.org/10.1145/3373376.3378512

[10] Lukasz Dudziak, Thomas Chau, Mohamed Abdelfattah, Royson Lee, Hyeji Kim, and Nicholas Lane. 2020. BRP-NAS: Prediction-based NAS using GCNs. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., 10480–10490. https://proceedings.neurips.cc/paper_files/paper/2020/file/768e78024aa8fdb9b8fe87be86f64745-Paper.pdf

[11] Vojislav Dukic, Rodrigo Bruno, Ankit Singla, and Gustavo Alonso. 2020. Photons: lambdas on a diet. In *SoCC '20: ACM Symposium on Cloud Computing, Virtual Event, USA, October 19-21, 2020*, Rodrigo Fonseca, Christina Delimitrou, and Beng Chin Ooi (Eds.). ACM, 45–59. https://doi.org/10.1145/3419111.3421297

[12] Tarek Elgamal. 2018. Costless: Optimizing Cost of Serverless Computing through Function Fusion and Placement. In *2018 IEEE/ACM Symposium on Edge Computing, SEC 2018, Seattle, WA, USA, October 25-27, 2018*. IEEE, 300–312. https://doi.org/10.1109/SEC.2018.00029

[13] .NET Foundation. 2023. IronPython. https://ironpython.net/.

[14] OpenJS Foundation. 2023. Worker threads | Node.js. https://nodejs.org/dist/latest-v18.x/docs/api/worker_threads.html.

[15] Python Software Foundation. 2020. Global Interpreter Lock. https://wiki.python.org/moin/GlobalInterpreterLock.

[16] Python Software Foundation. 2023. Cpython. https://github.com/python/cpython.

[17] Python Software Foundation. 2023. Jython. https://www.jython.org/.

[18] Python Software Foundation. 2023. ProcessPoolExecutor. https://docs.python.org/3.7/library/concurrent.futures.html?#concurrent.futures.ProcessPoolExecutor.

[19] Python Software Foundation. 2023. Python Multithreading without GIL. https://github.com/colesbury/nogil.

[20] The Apache Software Foundation. 2023. Apache OpenWhisk Composer is a new programming model for composing cloud functions built on Apache OpenWhisk. https://github.com/apache/openwhisk-composer.

[21] The PyTorch Foundation. 2023. LSTM. https://pytorch.org/docs/stable/generated/torch.nn.LSTM.html.

[22] Alexander Fuerst and Prateek Sharma. 2021. FaasCache: keeping serverless computing alive with greedy-dual caching. In *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*, Tim Sherwood, Emery D. Berger, and Christos Kozyrakis (Eds.). ACM, 386–400. https://doi.org/10.1145/3445814.3446757

[23] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. 2019. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*, Iris Bahar, Maurice Herlihy, Emmett Witchel, and Alvin R. Lebeck (Eds.). ACM, 3–18. https://doi.org/10.1145/3297858.3304013

[24] Sam Gross. 2023. PEP 703 – Making the Global Interpreter Lock Optional in CPython. https://peps.python.org/pep-0703/.

[25] Intel. 2018. Intel-64 and IA-32 architectures software developer's manual.

[26] Abhinav Jangda, Bobby Powers, Emery D. Berger, and Arjun Guha. 2019. Not So Fast: Analyzing the Performance of WebAssembly vs. Native Code. In *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*, Dahlia Malkhi and Dan Tsafrir (Eds.). USENIX Association, 107–120. https://www.usenix.org/conference/atc19/presentation/jangda

[27] Zhipeng Jia and Emmett Witchel. 2021. Nightcore: efficient and scalable serverless computing for latency-sensitive, interactive microservices. In *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*, Tim Sherwood, Emery D. Berger, and Christos Kozyrakis (Eds.). ACM, 152–166. https://doi.org/10.1145/3445814.3446701

[28] Brian W. Kernighan and Shen Lin. 1970. An efficient heuristic procedure for partitioning graphs. *Bell Syst. Tech. J.* 49, 2 (1970), 291–307. https://doi.org/10.1002/j.1538-7305.1970.tb01770.x

[29] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. 2018. Pocket: Elastic Ephemeral Storage for Serverless Analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, Andrea C. Arpaci-Dusseau and Geoff Voelker (Eds.). USENIX Association, 427–444. https://www.usenix.org/conference/osdi18/presentation/klimovic

[30] Swaroop Kotni, Ajay Nayak, Vinod Ganapathy, and Arkaprava Basu. 2021. Faastlane: Accelerating Function-as-a-Service Workflows. In *2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*, Irina Calciu and Geoff Kuenning (Eds.). USENIX Association, 805–820. https://www.usenix.org/conference/atc21/presentation/kotni

[31] Jie Li, Laiping Zhao, Yanan Yang, Kunlin Zhan, and Keqiu Li. 2022. Tetris: Memory-efficient Serverless Inference through Tensor Sharing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA. https://www.usenix.org/conference/atc22/presentation/li-jie

[32] 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 *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*, Babak Falsafi, Michael Ferdman, Shan Lu, and Thomas F. Wenisch (Eds.). ACM, 782–796. https://doi.org/10.1145/3503222.3507717

[33] Changyuan Lin and Hamzeh Khazaei. 2021. Modeling and Optimization of Performance and Cost of Serverless Applications. *IEEE Trans. Parallel Distributed Syst.* 32, 3 (2021), 615–632. https://doi.org/10.1109/TPDS.2020.3028841

[34] OpenFaaS Ltd. 2023. Classic Watchdog for OpenFaaS. https://github.com/openfaas/classic-watchdog.

[35] OpenFaaS Ltd. 2023. of-watchdog: Reverse proxy for STDIO and HTTP microservices. https://github.com/openfaas/of-watchdog.

[36] OpenFaaS Ltd. 2023. OpenFaaS: Serverless Functions, Made Simple. https://www.openfaas.com/.

[37] Ashraf Mahgoub, Karthick Shankar, Subrata Mitra, Ana Klimovic, Somali Chaterji, and Saurabh Bagchi. 2021. SONIC: Application-aware Data Passing for Chained Serverless Applications. In *2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*, Irina Calciu and Geoff Kuenning (Eds.). USENIX Association, 285–301. https://www.usenix.org/conference/atc21/presentation/mahgoub

[38] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Sameh Elnikety, Somali Chaterji, and Saurabh Bagchi. 2022. ORION and the Three Rights: Sizing, Bundling, and Prewarming for Serverless DAGs. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 303–320. https://www.usenix.org/conference/osdi22/presentation/mahgoub

[39] Microsoft. 2023. Durable Functions is an extension of Azure Functions that lets you write stateful functions in a serverless compute environment. https://docs.microsoft.com/en-us/azure/azure-functions/durable/.

[40] Inc MinIO. 2023. MinIO | High Performance, Kubernetes Native Object Storage. https://min.io/.

[41] Djob Mvondo, Mathieu Bacou, Kevin Nguetchouang, Lucien Ngale, Stéphane Pouget, Josiane Kouam, Renaud Lachaize, Jinho Hwang, Tim Wood, Daniel Hagimont, Noël De Palma, Bernabé Batchakui, and Alain Tchana. 2021. OFC: an opportunistic caching system for FaaS platforms. In *EuroSys '21: Sixteenth European Conference on Computer Systems, Online Event, United Kingdom, April 26-28, 2021*, Antonio Barbalace, Pramod Bhatotia, Lorenzo Alvisi, and Cristian Cadar (Eds.). ACM, 228–244. https://doi.org/10.1145/3447786.3456239

[42] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2018. SOCK: Rapid Task Provisioning with Serverless-Optimized Containers. In *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, Haryadi S. Gunawi and Benjamin Reed (Eds.). USENIX Association, 57–70. https://www.usenix.org/conference/atc18/presentation/oakes

[43] Linux Kernel Organization. 2023. CFS Scheduler. https://www.kernel.org/doc/html/latest/scheduler/sched-design-CFS.html.

[44] Pallets. 2023. Flask: The Python micro framework for building web applications. https://flask.palletsprojects.com/.

[45] Soyeon Park, Sangho Lee, Wen Xu, Hyungon Moon, and Taesoo Kim. 2019. libmpk: Software Abstraction for Intel Memory Protection Keys (Intel MPK). In *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*, Dahlia Malkhi and Dan Tsafrir (Eds.). USENIX Association, 241–254. https://www.usenix.org/conference/atc19/presentation/park-soyeon

[46] Daniel Barcelona Pons, Marc Sánchez Artigas, Gerard París, Pierre Sutra, and Pedro García López. 2019. On the FaaS Track: Building Stateful Distributed Applications with Serverless Architectures. In *Proceedings of the 20th International Middleware Conference, Middleware 2019, Davis, CA, USA, December 9-13, 2019*. ACM, 41–54. https://doi.org/10.1145/3361525.3361535

[47] Fission Project. 2023. Fission Workflows is a workflow-based serverless function composition framework built on top of the Fission. https://github.com/fission/fission-workflows.

[48] The PyPy Project. 2023. PyPy: Software Transactional Memory. https://doc.pypy.org/en/latest/stm.html.

[49] Datadog Research. 2020. The State of Serverless. https://www.datadoghq.com/state-of-serverless-2020/.

[50] Datadog Research. 2021. The State of Serverless. https://www.datadoghq.com/state-of-serverless-2021/.

[51] Giampaolo Rodola. 2023. psutil: Cross-platform lib for process and system monitoring in Python. https://psutil.readthedocs.io/.

[52] scikit-learn developers. 2023. RandomForestRegressor. https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestRegressor.html.

[53] Amazon Web Services. 2023. Amazon S3: Object storage built to retrieve any amount of data from anywhere. https://aws.amazon.com/s3/.

[54] Amazon Web Services. 2023. AWS Step Functions: Visual workflows for modern applications. https://aws.amazon.com/step-functions/.

[55] Amazon Web Services. 2023. Firecracker: Secure and fast microVMs for serverless computing. https://gvisor.dev/.

[56] Simon Shillaker and Peter R. Pietzuch. 2020. Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing. In *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*, Ada Gavrilovska and Erez Zadok (Eds.). USENIX Association, 419–433. https://www.usenix.org/conference/atc20/presentation/shillaker

[57] Eric Snow. 2023. PEP 684 – A Per-Interpreter GIL. https://peps.python.org/pep-0684/.

[58] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Joseph Gonzalez, Joseph M. Hellerstein, and Alexey Tumanov. 2020. Cloudburst: Stateful Functions-as-a-Service. *Proc. VLDB Endow.* 13, 11 (2020), 2438–2452. http://www.vldb.org/pvldb/vol13/p2438-sreekanti.pdf

[59] Akshitha Sriraman and Thomas F. Wenisch. 2018. μTune: Auto-Tuned Threading for OLDI Microservices. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, Andrea C. Arpaci-Dusseau and Geoff Voelker (Eds.). USENIX Association, 177–194. https://www.usenix.org/conference/osdi18/presentation/sriraman

[60] Mark Sutherland, Siddharth Gupta, Babak Falsafi, Virendra J. Marathe, Dionisios N. Pnevmatikatos, and Alexandros Daglis. 2020. The NEBULA RPC-Optimized Architecture. In *47th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2020, Valencia, Spain, May 30 - June 3, 2020*. IEEE, 199–212. https://doi.org/10.1109/ISCA45697.2020.00027

[61] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. 2021. Benchmarking, analysis, and optimization of serverless function snapshots. In *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*, Tim Sherwood, Emery D. Berger, and Christos Kozyrakis (Eds.). ACM, 559–572. https://doi.org/10.1145/3445814.3446714

[62] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. 1993. Efficient Software-Based Fault Isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating System Principles, SOSP 1993, The Grove Park Inn and Country Club, Asheville, North Carolina, USA, December 5-8, 1993*, Andrew P. Black and Barbara Liskov (Eds.). ACM, 203–216. https://doi.org/10.1145/168619.168635

[63] Xingda Wei, Fangming Lu, Tianxia Wang, Jinyu Gu, Yuhan Yang, Rong Chen, and Haibo Chen. 2023. No Provisioned Concurrency: Fast RDMA-codesigned Remote Fork for Serverless Computing. (2023), 497–517. https://www.usenix.org/conference/osdi23/presentation/wei-rdma

[64] Philipp A. Witte, Mathias Louboutin, Henryk Modzelewski, Charles Jones, James Selvage, and Felix J. Herrmann. 2020. An Event-Driven Approach to Serverless Seismic Imaging in the Cloud. *IEEE Trans. Parallel Distributed Syst.* 31, 9 (2020), 2032–2049. https://doi.org/10.1109/TPDS.2020.2982626

[65] Laiping Zhao, Yanan Yang, Yiming Li, Xian Zhou, and Keqiu Li. 2021. Understanding, predicting and scheduling serverless workloads under partial interference. In *SC '21: The International Conference for High Performance Computing, Networking, Storage and Analysis, St. Louis, Missouri, USA, November 14 - 19, 2021*, Bronis R. de Supinski, Mary W. Hall, and Todd Gamblin (Eds.). ACM, 22:1–22:15. https://doi.org/10.1145/3458817.3476215