ExDe: Design Space Exploration of Scheduler Architectures and Mechanisms for Serverless Data-processing

Sacheendra Talluri[a], Nikolas Herbst[b], Cristina Abad[c], Tiziano De Matteis[a], Alexandru Iosup[a]

[a] *Vrije Universiteit Amsterdam, Amsterdam, The Netherlands*
[b] *University of Würzburg, Würzburg, Germany*
[c] *Escuela Superior Politécnica del Litoral, Guayaquil, Ecuador*

Abstract

Serverless computing is increasingly used for data-processing applications in both science and business domains. At the core of serverless data-processing systems is the scheduler, which ensures dynamic decisions about task and data placement. Due to the variety of user, cluster, and workload properties, the design space for high-performance and cost-effective scheduling architectures and mechanisms is vast. The large design space is difficult to explore and characterize. To help the system designer disentangle this complexity, we present ExDe, a framework to systematically explore the design space of scheduling architectures and mechanisms. The framework includes a conceptual model and a simulator to assist in design space exploration.

We use the framework, and real-world workloads, to characterize the performance of three scheduling architectures and two mechanisms. Our framework is open-source software available on Zenodo.

1. Introduction

Scientific data analysis [1], business analytics [2], search-based decision-making [3], and other data-driven workloads with near-interactive deadlines require their computation to complete within short time spans. The short duration and the frequently changing quantity of resources required by these workloads [4] form a natural fit for serverless computing, which lets users lease only the resources required for a short period of time [5, 6, 7]. Both academia (e.g., funcX [1], Starling [8]) and industry (e.g., Snowflake [4], Databricks [9]) have recognized this synergy and have proposed systems to leverage it.

Serverless data-processing systems run their computation on distributed clusters of virtual machines or containers as nodes and use remote object storage (e.g., AWS S3, Azure Blob Storage). The data-processing applications are workflows composed of tasks. A task is a piece of computation that reads data, processes it, and writes out the intermediate or output result. The *scheduler* is a crucial component of these systems: it directs the tasks to different nodes in the cluster at the right time, based on resource requirements, locality-awareness, and other scheduling policies.

A scheduler is defined by its *architecture*, *mechanisms*, and the *actions* they enable. Architecture refers to how multiple schedulers coordinate to make resource management decisions [10]. Mechanisms refer to the set of all inter-component communication and bookkeeping that make policy enforcement possible [11]. The architecture and mechanisms enable the scheduler to enact a scheduling policy through actions. *Actions* are resource management and communication decisions like allocating resources, preempting tasks, and sending completion events that a scheduler can take to achieve policy goals.

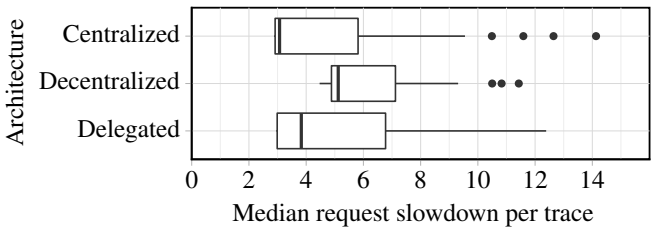


Figure 1: Distribution of median task slowdown (lower is better) per trace when using different scheduler architecture.

Scheduler architecture and mechanisms can significantly affect serverless system performance. Figure 1 shows the impact of the scheduler architecture on interactive data-processing task performance. The figure depicts the median task slowdown across multiple traces when using different architectures. The slowdown is the ratio of the actual task completion time to the ideal completion time. We observe that the decentralized architecture can result in 2× the amount of task slowdown than the centralized one. The traces are from the IBM COS dataset [12], each containing 3 million tasks, and simulated on a cluster sized to run each trace at 80% utilization. The cluster size ranges from 15 to 31 nodes depending on the trace.

Although much work focuses on policy (algorithm) design for scheduling, the design of the entire scheduler remains a key but underserved challenge, e.g., *Which architectures to leverage? Which mechanisms to include?* **Proper exploration of scheduling design space is challenging.** Changing the architecture or adding new mechanisms requires time-consuming engineering efforts, as in the case of Condor [13] and Borg [14], making design space exploration expensive. Often the mecha-

Email address: s.talluri@vu.nl (Sacheendra Talluri)

nisms, and the actions they enable, are under-specified and implicit in the system design. The implicitness makes isolating and modifying them difficult, rendering the systematic exploration hard [15, 16]. Finally, we observe that not all actions in the scheduling reference architecture [15, 17] are available to all schedulers [18, 19]. While some actions can be implemented by changes to local bookkeeping mechanisms in the scheduling policy, many cannot, depauperating exploration options.

In this work, we address these challenges by proposing ExDe, a framework to systematically and conveniently explore the design space of scheduler architectures and mechanisms for serverless computing.

With ExDe, we introduce the concept of a *scheduler frame* [1] to encompass all such actions and the architecture and mechanisms that enable them.

> Formally, we define the **scheduler frame** as the set of all mechanisms in the scheduler that enable actions not possible by any local modification of the scheduler algorithm and policy. Instead, a frame requires coordination between multiple scheduler components.

The host software (e.g., hypervisor, kubelet [20]), the broker, and the data manager (e.g., Pocket [21]) are examples of scheduler components. The components are described further in Section 2. Hierarchical scheduling and work stealing are examples of mechanisms that constitute a scheduler frame.

ExDe uses scheduler frames to allow system designers to explicitly define and explore actions that require the coordination of multiple scheduler components to occur. It comes with various scheduler frames ready to be used and modeled after the state-of-the-art alternatives available for each design choice. ExDe relies on trace-based discrete event simulation [22]. We justify our design choice with the following observations. First, existing serverless data-processing systems perform other tasks besides scheduling [23, 1], and it can be challenging to isolate the impact of the scheduler frame. Using simulation, the user can evaluate different frames in isolation. Second, many existing approaches use a limited set of workloads or individual applications in their evaluation as it is expensive and time-consuming to evaluate each system on a wide variety of workloads [24]. By resorting to simulation, we conduct thousands of performance evaluations in a timely and cost-effective manner.

The use of simulation presents a challenge in terms of validation. To overcome this challenge, we adopt a two-step approach. First, we utilize real-world measurements in our network model, which makes our simulations more closely approximate the real-world [25, 26, 27]. Second, we transform our simulation model into an emulated system implementation, conduct experiments, and compare the results from the two systems.

Towards improving the design space exploration process for serverless data-processing system scheduler design, we make a three-fold contribution:

[1] The term is inspired by the structural frame used in construction. The building's frame cannot change, but the layout and composition of the floors can. The word framework shares the same etymological roots.

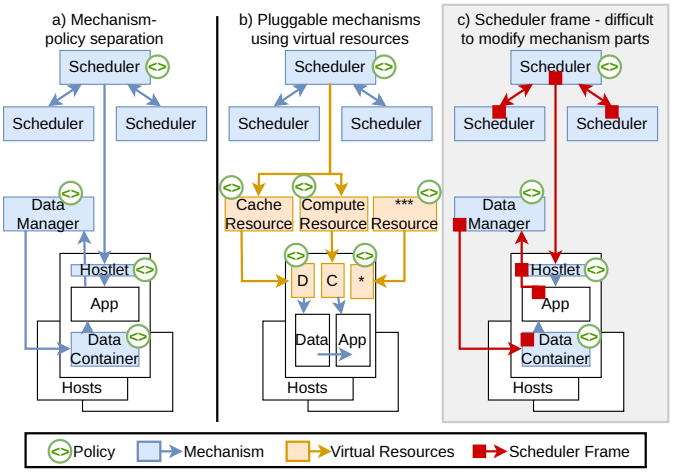


Figure 2: The scheduler frame model differentiated from the mechanism-policy separation and pluggable mechanism conceptual models. The scheduler frame consists of the parts of the systems which require multiple components to be changed to enable new features.

1. We design a conceptual model of scheduler frames (Section 2). We classify a number of existing scheduler designs into design frames. The designs we classify include different scheduling architectures and mechanisms.

2. We design, implement, and validate a trace-driven simulator to enable design space exploration of scheduler frames for serverless data-processing systems (Section 3). Our simulator provides first-class support for inter-component communication enabling ergonomic scheduler frame implementation.

3. We conduct systematic design space exploration using ExDe (Section 4). We evaluate representative design frames related to scheduler architecture, work stealing, and data migration across different cluster and workload configurations, considering a set of 54 real-world traces from IBM [12].

We make ExDe open-source and easily extensible to enable practitioners to re-run the experiment as storage and network characteristics change in the future. It is available at `https://zenodo.org/record/7829151`. Our approach ensures that the simulations accurately reflect real-world scenarios and can be applied in practical computer systems.

2. Conceptual Model of Scheduler Frames

In this section, we describe the motivation behind the conceptual model, expand on the definition of the scheduler frame, describe scheduler components, introduce how frames encompass mechanisms, and how the conceptual model can help scheduler design.

The flexibility in scheduler design makes it difficult for the system designer to qualitatively and quantitatively compare different scheduler designs. The community has tackled this problem for a long time and has adopted mechanism and policy

Table 1: The components affected and actions made available by different mechanisms. Actions are changes to system state that the mechanisms enable. (DM = Data manager)

Mechanism	Components used						Implementations	Frame actions (Not exhaustive)
	Placer(s)	Broker	Host	Client	Metadata	DM		
Architecture	✓	✓		✓			Centralized [28, 29], decentralized [30, 4], delegated [31, 13], hybrid [32, 33]	Pick scheduler, read metrics
Preemption	✓		✓				Threshold-based [34], fair sharing [24]	Migrate
Control-flow	✓	✓	✓				Push/pull [35], speculative exec. [36]	Pull/push task, rollback state
Data placement	✓				✓	✓	Shuffle [27], intermediate data [21]	Store, move, etc.
Fault tolerance	✓			✓	✓	✓	Checkpoint [37], retry [38]	Recover, retry
Networking	✓				✓		NetHint [39]	Change placement
Barriers	✓	✓					Gang scheduling [18, 40]	Reserve hosts

split as a standard practice [11] (Figure 2a). All the components, subsystems (e.g.: containers, caches), and communication mechanisms which enable the policy to manage resources are the mechanisms. The policy itself can be split into the optimization goal we wish to achieve and the algorithm we use for optimization [41]. However, the comparison and systematization of the mechanism have received less attention than the algorithms [42, 43] and the policies [44, 45].

Mechanisms which can be plugged into the system during runtime were introduced in OS design in the 1980s with virtual resources [46]. Pluggable mechanisms (Figure 2b) have been used to enable programmable caches [47] and scheduler mechanisms such as gang scheduling [18]. In Figure 2b, virtual resources such as the cache are used by the scheduler to schedule tasks. These virtual resources in-turn are scheduled onto the physical resources. The scheduler can have virtual resource specific scheduling policies. For example, the cache resource can have a corresponding eviction policy which is irrelevant to other resources.

We present the conceptual model of scheduler frames to help the system designer tackle the difficulty of mechanism design. Comparing different alternative designs and assessing their fit for the requirements is integral to the design process. Using scheduler frames, the designer can compare if two different scheduler designs can support the same mechanisms. If they cannot, the designer can qualitatively compare the effort of implementing the necessary features to support additional mechanisms. This comparison is possible because a scheduler frame can explicate the actions a design needs to support the desired mechanisms.

2.1. Components of a Scheduler Frame

We depict an example scheduler model with the scheduler frame highlighted in Figure 2c. The scheduler is composed of multiple components. The components interact to provide mechanisms related to workflow scheduling, data storage, speculative execution, and more. The figure depicts the frame overlaid on top of the interacting components. The *scheduler frame* provides common features and the data model used by the components to coordinate with each other. Once a frame is designed and implemented, components can use it to implement mechanisms.

We describe briefly components in used Figure 2 and Table 1 here.

Placer (Scheduler): Placer is the part of the scheduler that assigns application tasks to appropriate resources. In Figure 2, it is a part of the scheduler. A scheduler can have multiple placers. In a fully decentralized scheduler, placers might not even interact with each other or only interact occasionally.

Broker (Scheduler): In some distributed schedulers, placers interact via a centralized component called the broker. The broker decides which subset of the resources each placer is responsible for.

Host software: The host software is responsible for interacting with the scheduler enforcing the scheduler's decisions on the host. Kubelet [20] in Kubernetes and different hypervisors are examples of host software. Host software limits the actions available to the scheduler. For example, a scheduler cannot implement migration or preemption without host software which support that feature. The hostlet in Figure 2 represents the host software.

Client: The client uses the scheduler to schedule applications. It can provide the scheduler with relevant information for better scheduling. For example, a client can inform the scheduler of data objects the application accesses so that the application can be scheduled closer to the data location.

Metadata: Different scheduler components use the metadata storage component to store resource state and application state. The Kubernetes project uses etcd for this purpose. For example, if an application needs access to a particular system service, the metadata helps locate that service using its service discovery feature.

Data manager: Application data needs to be tracked, replicated, and secured even when applications are not using the compute resources. The data manager keeps inventory of the data objects on each physical node. It moves and replicates the data objects to balance load.

2.2. Scheduler Frames in Existing Systems

Examples of scheduler frames include communication protocols for interacting between scheduler components, the consistency models the components use to agree upon data correctness, coordination mechanisms for migration and rollback,

etc. The Kubernetes data model is an example of *communication protocols* which form the frame. Users can implement new mechanisms in Kubernetes through operators [48]. The operators need to make use of the Kubernetes data API to interact with other Kubernetes components and operators. Even a basic action such as starting a container requires using the API. The *consistency models* used in Omega [49] are another example of a scheduler frame. Omega explores different consistency models for distributed schedulers. For a scheduler to effectively enforce its scheduling policies, all components of a distributed scheduler need to adhere to the same consistency policy.

The *architecture* involves implementing the appropriate mechanisms in the placers, the broker, and the client. Even a decentralized architecture where the placers do not communicate with each other could use a broker to coordinate global actions such as node addition and removal. The *control-flow* mechanism requires the implementation in host, placer, and broker. Work stealing requires control-flow support for hosts to steal tasks from other hosts via the placer. Speculative execution requires host and place modification [36]. Speculative execution also requires support from the data manager to read not yet committed data, commit data of successful speculative executions and rollback stored data from failed speculation. The *data placement* mechanism uses the metadata to estimate the load and object popularity across different hosts and uses the data manager to migrate data to improve load balance.

Multiple components need to coordinate and work together for the mechanisms to function. This is the essence of a scheduler frame. A policy implemented in just one component, such as the placer or the host, cannot implement these mechanisms by itself. The policy is limited in the actions it can perform by the frame. For example, the host cannot implement pull-based work stealing alone. The placer must support hosts pulling tasks and remove stolen tasks from other host queues.

We summarize some mechanisms, systems that implement them, and the frames requires by them in Table 1. We mark each component that is modified by one of the systems with a checkmark (✓). We list the actions made available to the scheduling policy by each mechanism. Actions are changes to system state that the mechanisms enable.

Using the scheduler frames concept, system designers can systematically reason about the actions that will be available to implement scheduling mechanisms and policies. They can then evaluate the engineering cost of implementing the mechanism and the performance (throughput, cost, energy, or other metrics) benefit of making the corresponding actions available.

3. Simulation-based Process for Scheduler Frame Characterization

Evaluating the performance of a scheduler frame requires characterizing the mechanisms that utilize the frame. Implementing all the mechanisms in a real system can be onerous and cost a lot of engineering resources. Therefore, we turn to simulation to characterize and explore different scheduler frames.

Figure 3 depicts the process a system designer would use to evaluate a scheduler frame using simulation. We start with a set

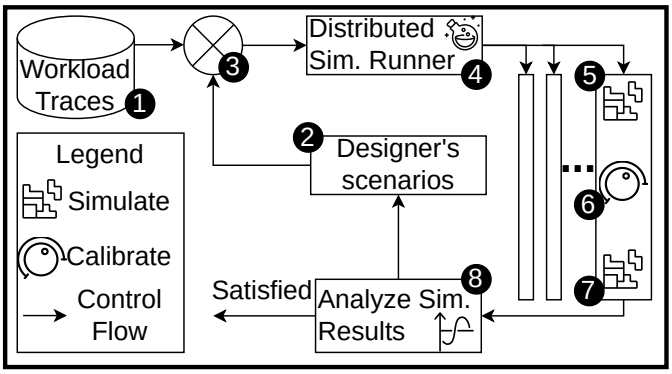


Figure 3: The process to characterize a scheduler frame.

Table 2: Mechanisms implemented in the `ExDe` simulator.

ExDe additions	Already in OpenDC
Architecture-support	Simulator core
Work stealing	Scheduling policies
Locality-awareness	Preemption
Caching	Trace readers
I/O latency models	Energy models

of *workload traces* ❶. The traces are combined ❸ with *scenarios* ❷ the designer wants to characterize to come up with a set of simulations to run. The *distributed simulation runner* ❹ executes the simulations on a cluster of machines. Each simulation run consists of three phases. The *simulator* ❺ is first run with an approximate set of parameters. The simulator parameters are then *calibrated* ❻ using the results of the previous simulation to meet system utilization constraints. The calibration is necessary because different traces require different sized cluster to achieve the same utilization. Finally, the simulation is run again with calibrated parameters ❼. The simulation results are then analyzed by the designer using distributed *analysis tools* ❽ we provide, or the designer herself develops. The designer uses the analysis results to configure the scenarios for the next round of the process. The process is repeated until the designer is satisfied with the results.

We use the Ray [50] framework to implement our distributed simulation runner. The designer supplies simulation parameters as rows in a Pandas [51] dataframe. The rows are then converted to command-line arguments that are passed on to each individual OpenDC simulator run. We also use Ray and Pandas to implement the tools to analyze simulation results.

3.1. Implementing Scheduler Frames in the Simulator

We extend the OpenDC simulator [22], implementing a framework to support scheduler frame characterization. The framework consists of easily-extensible components and mechanisms that can be combined to make a scheduler frame. The simulator has constructs to ergonomically model communication between components. Both the simulator and `ExDe` are implemented in the Kotlin programming language.

```
interface SchedulerFrame {

    // Metadata components
    val keyToHostMap: Map<Key,Host> (1)
    val perKeyScore: Map<Key,Double>

    // Blocking queues
    val hostQueues: Map<Host,BQueue> (2)
    val globalQueue: BQueue

    fun getNextTask(h: Host): Task? { (4)
      val hostQ = hostQueues[h]
      val task? = select { (3)
        hostQ, globalQueue
      } ...
    }

    fun offerTask(t: Task) { (5)
      ...
      chosenQueue.add(t)
    }
}
```

Figure 4: SchedulerFrame Interface Outline

The user can easily define a new scheduler frame by implementing the *SchedulerFrame* interface depicted in Listing **??**. The interface offers *metadata storage* ❶ components required by mechanisms such as data placement and networking. A set of *blocking queues* ❷ is used to represent asynchronous communications between scheduler components. Each host server has a queue, but other queues can be created based on mechanism needs. The *select* ❸ functionality allows a component to implement multiple strategies for receiving asynchronous communication from other components. The blocking queues are not FIFO, but can be used like them. Users can inspect all tasks in the queue, sample tasks, or specify priorities to implement custom scheduling policies. A task can be dequeued from any position in the queue. Apart from these policies, there are local scheduling policies unrelated to communication that are built into OpenDC.

The *getNextTask* ❹ method is used by the hosts to retrieve the next task to run. It supports both push-based and pull-based scheduling. The *offerTask* ❺ method is used by clients to enqueue tasks for scheduling. All components required to implement a scheduler are injected, and their interactions are specified in the scheduler frame. The scheduler frame is also responsible for transferring the task objects to the required components. All components are provided with a virtual clock which they can advance on a per-task basis. The time duration a task spends using a resource is sampled from a user-specified empirical model [52], linear model [27], or stochastic model [53] (e.g., Pareto distribution).

The scheduler frame interface abstracts away numerous details necessary for simulation, such as reading the workload trace, writing the results, warming up the simulator for steady-state execution, the host model, the network model, and more. Table 2 lists all the components and mechanisms we implemented in ExDe and the ones already available through OpenDC.

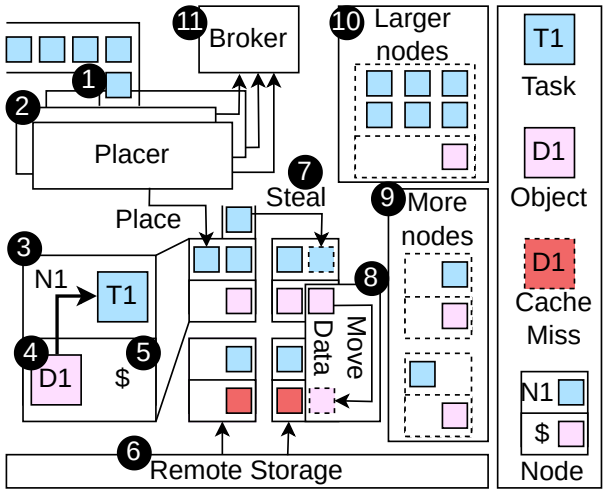


Figure 5: Serverless data-processing system model with possible design decisions.

With this approach, we enable users to specify accurate storage and network models to enable the exploration of different cloud computing paradigms, from virtual machines to serverless computing.

4. Using ExDe to Characterize Frames for Serverless Data-processing

We use ExDe to characterize the impact of different scheduler frames on serverless data-processing systems. The model of the system we characterize is depicted in Figure 5. Data-processing applications are structured as workflows of *tasks* ❶. The *placer* ❷ assigns the tasks to *nodes* ❸. Tasks read *data objects* ❹ they process from *local cache* ❺ or *remote storage* ❻.

In our scheduler frame, we consider the following properties invariant: *i)* the local cache is partitioned across nodes; and *ii)* the placer is locality-aware and tries to schedule tasks in a locality-aware manner. Locality-aware scheduling means that the scheduler knows in which node an object is likely to be cached and directs tasks accessing that object accordingly.

The design space that we consider for inclusion in our scheduler frame consists of the *scheduler architecture* (centralized, decentralized, or delegated), *work stealing* ❼, and *data migration* ❽.

We characterize the impact of this design space across different *cluster sizes* ❾ and *node sizes* ❿. We also characterize the impact of cluster scaling (size change) and workload properties like the popularity distribution. Next, we describe in detail the design space outlined above.

Scheduler Architecture. The scheduler architecture affects engineering complexity, performance (task execution latency), and

scalability (number of tasks it can timely schedule as they arrive). We consider three design alternatives: Centralized, decentralized (ring hash), and delegated.

A centralized scheduler consists of a single placer and is the easiest to implement correctly, but can not increase scheduling throughput by adding more placers. When the workload requires more throughput than can be provided by a centralized scheduler, one can choose between a decentralized scheduler architecture (e.g.: Snowflake-like [4]), or a delegated scheduler architecture (e.g.: Atoll-like [31]) with multiple centralized placers coordinating using a *broker* (11). Out of these two designs, the decentralized one is easier to engineer than the delegated one as it requires no coordination between the nodes.

All schedulers we implement use a locality-aware greedy scheduling policy. Locality-awareness means that the scheduler tries to assign tasks to a node where the data object required by the task is already available. The centralized scheduler is aware of the location of all data objects and the load of all nodes. So, it schedules a task to a node with the required object, and the least loaded node if the object is not cached. The delegated scheduler has multiple placers which assign tasks to nodes. The fraction of nodes managed by each placer are dynamic and depend on the node of the placer. The node to placer allocation is balanced every 5 minutes of simulation time. The decentralized scheduler uses a consistent hash to always schedule a task requiring a certain object to the same node. The object identifier is used as input to the hash function.

The centralized scheduler is aware of the load of all nodes. In the delegated case, the placers are aware of the load of the nodes assigned to them. The broker periodically reallocates the nodes as it is aware of the load of each placer. The decentralized scheduler is not load aware. A decentralized load-aware version is possible by using two different hashes and trying power of two random choices between the output. But, it would still be unaware of the load of all other nodes.

Schedulers queue tasks at the worker nodes. When multiple placers in the decentralized scheduler queue tasks at the same worker node, they are added to the queue. The queue is processed in a FIFO order.

Work Stealing. When work stealing is supported by the scheduler, a node with execute capacity can pull a task from another node's queue (e.g., from the node with the largest wait queue) and execute that task. This allows the nodes to correct scheduling decisions that have led to the overload of some workers while some others have spare capacity. This technique leads to reduced task latency at the cost of extra scheduling logic. Unlike traditional workload stealing, in locality-aware scheduling the issue of heterogeneous task execution latency is an issue: the stealing node can execute the task sooner, but the task may take longer to execute. The increased latency is because the stealing node does not have the task's data in the cache, but the original node likely does.

In a centralized scheduler, whenever a node doesn't have any tasks in the queue, the centralized scheduler picks a task from the busiest node to assign to it. In the delegated scheduler, the empty node contacts two other schedulers and picks a task to execute from the busiest one. In the decentralized case, the empty node also contacts two random schedulers and picks a task from the busiest one.

Table 3: Overview of experiments.

Section	Varying parameters	Fixed parameters	Metrics
§ 5.1	Cluster size (utilization)	Workload, Tasks per node = 4	Slowdown, storage delay, wait time
§ 5.3	Tasks per node	Workload, Utilization = 0.8	Slowdown
§ 5.4	Workload (doubles in size after half time)	Utilization = 0.8, Tasks per node = 4	Slowdown, Resource usage
§ 5.5	Workload (Object popularity distribution)	Utilization = 0.8, Tasks per node = 4	Slowdown

Data Migration. When data migration is supported by the scheduler, data objects are migrated from busy nodes to idle nodes based on a trigger. We evaluate three triggers in this work. The *global* trigger periodically migrates the most popular objects from busy nodes to idle nodes till load balance is achieved. The popularity of an object is computed based on the number of tasks that read the object. The *per task* trigger migrates a task based on the time since insertion or last migration. On timer expiration, the scheduler checks if a node with a lower load than the object's current node and migrates the object. The *work-steal* trigger can only be used when work stealing is enabled. When a task is stolen, it moves the object the task is accessing from the original node to the stealing node. Data migration can lead to increased storage delay if triggered too frequently. But, with the appropriate policy, it can even out load imbalances.

4.1. Experiment Setup

We run a series of experiments using our simulator to characterize the impact of the aforementioned design decisions on scheduler performance. We choose a realistic network model for the simulation based on real-world measurements. For communication with remote storage, we use an empirical distribution with bandwidth and latency values from recent work [25, 26]. According to Bian and Ailamaki [25], the median latency to remote storage (AWS S3) is 13 ms and the tail latency can exceed 1,000 ms. The average read bandwidth is 80 MBps [54]. We set the latency for communication between components to 1 ms based on recent characterization of networking between virtual machines [52, 55]. The latency and bandwidth distributions can be changed by the user, and the characterization rerun, to update the results for the changing ecosystem. We use the FIFO scheduling policy for all experiments. In the case of decentralized and delegated architectures, the policy is local FIFO per placer.

For our simulation, we use 54 real-world traces of data access patterns from the IBM COS dataset [12]. We use the traces

to generate a simulable trace of task executions. Each simulable trace consists of 3 million tasks executed over three hours. The median task runtime is 100 milliseconds. In total, we run 10,230 simulations for this characterization. Each simulation ran for approximately 1.5 minutes. All experiments consumed over 255 hours of CPU time. Including the exploratory experiments, and other experiments not analyzed in the paper, the simulations consumed over 1,000 hours of CPU time. As running many simulations is an embarrassingly parallel problem, we parallelized the evaluation over a cluster of 20 physical machines.

The experiments we run to characterize the design are outlined in Table 3. We characterize the impact of architecture, work stealing, and data migration as a function of the varying parameters. All parameters are configurable and easy to change using command-line arguments. This allows users to quickly rerun the characterization, and reevaluate their design decisions with a different set of parameters as their cluster setup changes.

5. Frame Characterization Results

Using simulation, we characterize the performance of different decisions that are part of a scheduler frame across different cluster configurations (node utilization, node size, and scaling). Table 3 summarizes the different experiment configurations.

We quantify system performance using the *task slowdown* metric, defined as the ratio of the actual execution time of a task over its ideal execution time. Task slowdown is a ratio and has no units. Because we use multiple traces, and each trace has its task slowdown distribution, we analyze them using summary metrics representing their distribution. We use the median and the tail latency (99th percentile) of the task slowdown distribution for each trace. In the following, we refer to them as the `tracemed` and `tracetail`, respectively. We quantify both the median and the dispersion between the 25th and 75th percentile values of the summary metrics. The dispersion between the 75th percentile and 25th percentile values is the Inter Quartile Range (IQR)

The actual execution time of a task is the sum of the time spent processing, waiting in the scheduler queue (*wait delay*), and waiting for remote storage (*storage delay*). A high wait delay is caused by a load imbalance across cluster nodes in systems running below saturation (100% utilization) for most of the experiment duration [56]. All our experiments meet this criterion. Hence, we use wait delay as a proxy for the amount of load imbalance in the system. The storage delay is a proxy for the number of tasks experiencing cache misses and accessing remote storage. We further investigate the root cause of the slowdown in-depth for a specific experiment in Section 5.2.

5.1. Impact of Utilization-level

One cluster configuration parameter available to the system designer is the number of nodes in the cluster (cluster size). For the same workload, a lower number of nodes implies higher system utilization. It is important for the system designer to know which scheduler frames can help her meet performance objectives at a given utilization. In this section, we investigate how and why the performance of different scheduling frames varies with cluster utilization using simulation. Cluster utilization is the ratio of time the cluster resources are being used to the total time they are available. The cluster size used in the simulation is different for different traces. For results to be comparable, the different traces are simulated in configurations such that the average utilization for each simulation is the same. Fixing the utilization required us to vary the cluster size per trace. The specific cluster size for a simulation is determined in the calibration phase specified in Section 3. The cluster size ranges from 15 to 31 nodes.

We consider four scenarios. In scenario À we compare the three considered scheduler architectures. In scenario Á we consider the same frames but with the work stealing mechanism enabled. Each frame has an architecture-specific work stealing implementation. Scenario Â compares two data migration policies for two frames. We only choose two because the fully decentralized frame does not support the central coordination required for data migration. Finally, in scenario Ã we quantify data migration combined with work stealing.

The `tracemed` and `tracetail` for all four scenarios considered are depicted in Figure 6. For each scenario, the top plot depicts the median slowdown per trace (`tracemed`), and the bottom plot shows the tail slowdown (`tracetail`) per trace. The horizontal axis in each figure shows the system utilization, and the vertical axis the task slowdown per trace. The lines in the plot depict the trend of median `tracemed` and `tracetail`. The points in the plot are the median measurements. The whiskers around the median indicate the IQR. The points and whiskers have minor horizontal adjustments to improve clarity.

Considering scenario À, the centralized scheduler exhibits the lowest median `tracemed` and the decentralized scheduler the highest (worst) at all utilization levels. The `tracemed` distribution exhibits high IQR for all architectures at high utilization (0.8). The centralized architecture also exhibits the lowest median `tracetail` for all utilization levels up to 0.7. All architectures exhibit high `tracetail` IQR at all utilizations. The decentralized architecture exhibits an order of magnitude worse `tracetail` performance than the delegated and the centralized architectures.

Next, in scenario Á we look at the impact of the work stealing operational technique on all architectures. In this case, the median `tracemed` increases from 2 to 2.5 with work stealing for the centralized and delegated architectures at utilizations below 0.6. The decentralized architecture has the least `tracemed`. The `tracemed` IQR at 0.8 utilization decreases significantly for all architectures with work stealing. Work stealing significantly impacts the `tracetail` distribution of all architectures. The median `tracetail` decreases from 10 to 5 for the centralized and decentralized architectures at low utilization (0.5). At high utilization (0.8), the median `tracetail` decreases an order of magnitude for all architectures.

In scenario Â, we quantify the slowdown of two different data migration policies. The two policies, global and per task, are described in Section 4. We observe that the migration policy

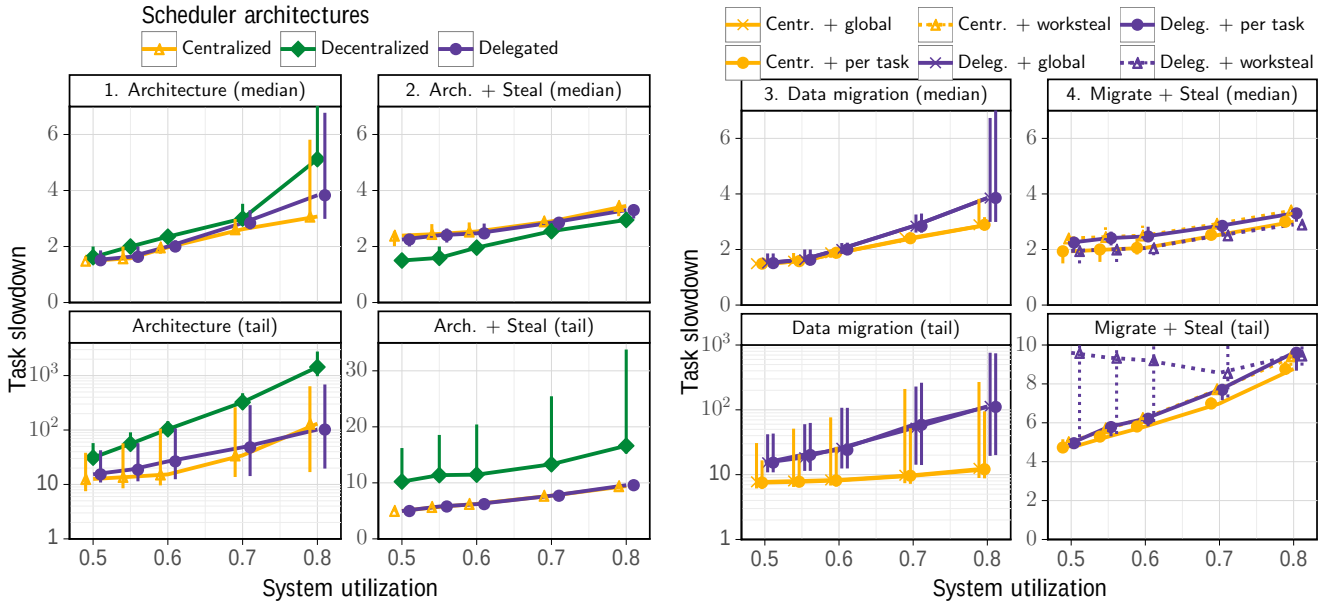


Figure 6: The impact of scheduler architecture and operational techniques on performance, represented by the task slowdown per trace, at different system utilization levels. The lower, the better. Horizontal axes are normalized, while vertical axes use different scales to appreciate differences. The points and whiskers have minor horizontal offset to improve clarity. The `tracemed` and `tracetail` of scenarios À, Á, Â, and Ã are represented.

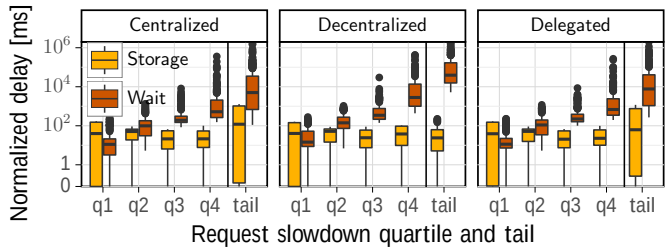


Figure 7: Normalized delay contribution from wait delay and storage delay for different architectures. Lower is better.

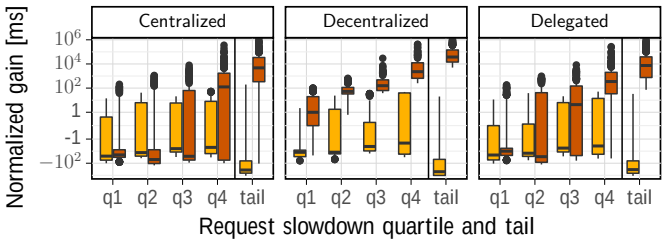


Figure 8: Normalized delay reduction (gain) because of work stealing. Higher is better.

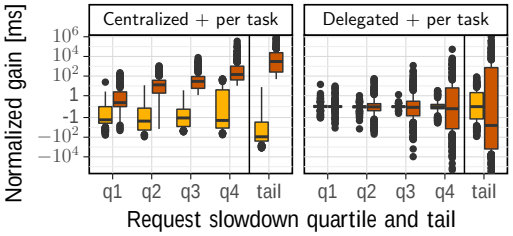


Figure 9: Normalized delay reduction (gain) because of data migration. Higher is better.

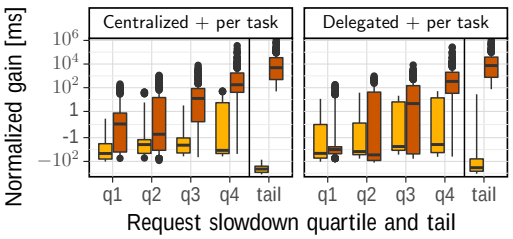


Figure 10: Normalized delay reduction (gain) because of data migration and work stealing. Higher is better.

has no effect on the `tracemed`. At high utilization ($\geq$ 0.7), data migration reduces the median `tracetail` by an order of magnitude for centralized and delegated architectures. But, it has no effect on the `tracetail` IQR.

In scenario Ã, we quantify the slowdown of combining work stealing with data migration. We term this combination mechanism `migsteal` for ease of reference. We use two migration policies, per task and worksteal, to represent two ways to combine work stealing with data migration. For per task, we combined the per task data migration policy (similar to Â) with a work stealer. For worksteal, we trigger data migration on a work steal event. We observe that combining work stealing with migration does not reduce slowdown any better than just using work stealing.

5.2. Slowdown Attribution

In this section, we analyze the root causes for all four scenarios from Section 5.1 of the slowdown for one experiment configuration. The configuration we use is the 0.8 utilization one. We investigate by attributing the delay experienced by a

task, over its ideal runtime, into wait delay and storage delay. We find the cause for both the median value of `tracemed` and `tracetail`, and the dispersion. This attribution helps us understand how well the different frames balance load among cluster nodes. It also helps us understand how the different frames navigate the trade-off between better load balance and more remote storage reads.

Figure 7 depicts the normalized delay distribution across all workload traces for the three frames involving only the architecture. The distribution is depicted for five categories (four quartiles and the tail) for each frame. The distribution across all workloads is depicted using boxes and whiskers. The box and whiskers have their usual meaning. The delay is normalized by dividing the cumulative delay in each category (quartile/tail) by the number of tasks in that category. We need to normalize at the tail category has much fewer tasks than the quartile categories. Normalizing per task makes the tail comparable to the other categories. We plot the individual quartiles instead of the CDF as each quartile has measurements for multiple traces. A combined CDF for all traces would be unreadable.

Figures 8, 9, and 10 depict the reduction (gain) in delay due to the frame including mechanisms in addition to the architecture. The gain is also normalized. The gain is obtained by subtracting the delay experienced when using mechanism from the delay experienced by the architectures without any mechanisms. A positive gain is an improvement. A negative gain means the delay worsened. *Note that all plots have a log scale.*

Considering scenario À (Figure 7), we observe that the wait time, and therefore imbalance, is the leading cause of slowdown at the higher quartiles. We observe that the storage delay has low dispersion, while the wait delay has high dispersion q3, q4, and the tail. From this, we can attribute the high IQR we observed for the `tracemed` and the `tracetail` to the wait delay. We observe that the decentralized architecture experiences an order of magnitude higher tail delays than the centralized or delegated architectures. A high wait delay is caused by imbalance. Imbalance causes head of line blocking on the busy nodes leading to high wait times. The decentralized architecture has high imbalance because each placer in our implementation of the decentralized architecture independently makes decisions, either using consistent hashing, which is sometimes combined with power of two random choices. Therefore, the placer is not aware of the load across the cluster and only aware of the node across two nodes in the case of two random choices.

High wait delay can also be caused by complex scheduling policies and insufficient scheduler resources. We use a simple scheduler policy and allocate plenty of resources to the scheduler to control for these factors in our experiments. We do not further explore the impact of these factors. But a different scheduler policy can result in different findings from what we obtain.

In scenario Á (Figure 8), we observe that the decentralized architecture experienced the most gain due to work stealing. The decentralized architecture experiences gain in all quartiles. This implies that even lower quartile tasks experience blocking with the decentralized scheduler. The centralized architecture and the delegated architectures also experience gains in the higher quartiles. The negative gain for the centralized scheduler in the lower quartiles explain the increase in median `tracemed` from 2.5 to 3. Work stealing reduces head of line blocking at the expense of median latency for the centralized architecture.

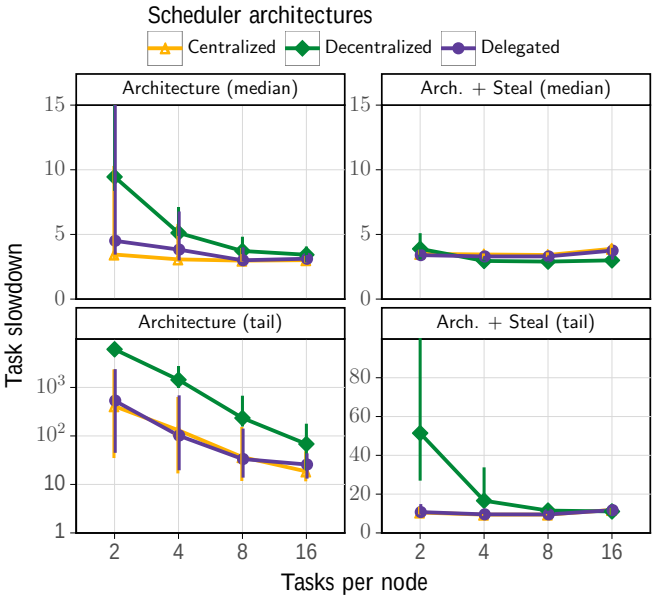


Figure 11: Impact of node size on task slowdown.

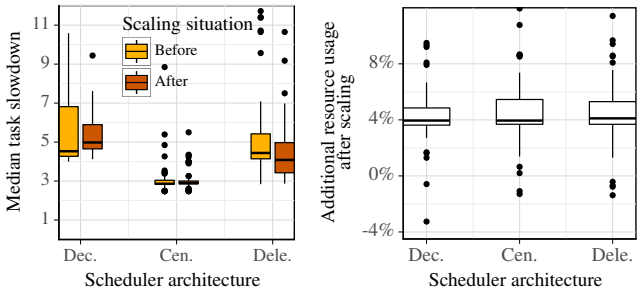
In scenario Â (Figure 9), we observe that the centralized architecture benefits from multiple orders of magnitude higher gain than the delegated architecture. The gain is highest in the tail. This indicates that data migration reduces load imbalance, and hence head of line blocking for the centralized architecture. The median gain for the delegated architecture is negative, but we see high 75th percentile gains in the tail. This implies that data migration combined with the delegated scheduler reduces blocking for some traces, but not others.

In scenario Ã (Figure 10), we observe that work stealing combined with data migration results in gains across all quartiles for both the centralized and decentralized architectures. The gain for the delegated architecture is similar to using only work stealing (Figure 8). The centralized architectures see higher gain in the lower quartiles compared to just using work stealing.

Key takeaways: All architectures exhibit similar performance at low utilizations. Work stealing improves tail performance by an order of magnitude in all cases. Data migration is only beneficial with the centralized architecture. The decentralized architecture, while simple to implement and horizontally scalable, comes at a performance cost. The performance cost is greatly reduced by work stealing. The delegated architecture approaches the performance of the centralized architecture with work stealing, but requires more implementation effort.

5.3. Impact of Node Size

The node size refers to the number of tasks a node can process simultaneously. The system designer can choose to populate her cluster with a few large nodes instead of many small nodes. In this experiment, we evaluate the performance spectrum between these two extreme scenarios. We evaluate the



(a) Performance impact. (b) Additional resource usage.

Figure 12: Impact of scaling on task slowdown.

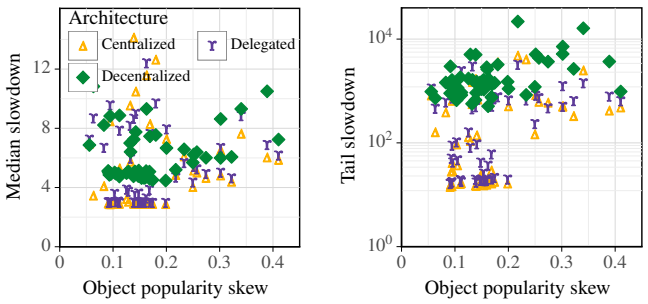


Figure 13: Impact of object popularity distribution on task slowdown.

performance of cluster configurations ranging from many small nodes with 2 processing slots each, all the way to a configuration with a few large nodes with 16 processing slots each. The total number of resources (task processing slots) in the cluster and the cluster utilization (0.8) remain the same throughout the experiment. Similar to Section 5.1, the cluster size used in the simulation is different for different traces to ensure that the utilization is the same for all traces. The specific cluster size for a simulation is determined in the calibration phase specified in Section 3. The cluster size ranges from 7, in a scenario with 16 processing slots per node, to 60, in a scenario with 2 processing slots per node.

Figure 11 depicts the results of our evaluation under two scenarios: with and without work stealing. The horizontal axis represent the node size in number of tasks a node can process. The vertical axis represents the `tracemed` and `tracetail`. We observe that the `tracemed` decreases for all architectures with increasing node size. The `tracetail` also decreases over an order of magnitude for centralized and delegated architectures with increasing node size. It decreases two orders of magnitude for the decentralized architecture Work stealing can result in low `tracemed` and `tracetail` even at small node sizes. At large node sizes, all architectures achieve performance close to work stealing, but without the engineering effort of implementing work stealing.

Key takeaway: Low slowdown, comparable to that achieved with work stealing, can be achieved by using a small-sized cluster with large nodes.

5.4. Impact of Scaling

Changing the number of nodes in a cluster dynamically while it is in operation, using autoscaling, is a common operation in serverless data-processing clusters. We investigate if different architectures impact the task performance after the dynamic scaling operation. We observed similar results for doubling the workload and halving the workload. Therefore, we only present results for doubling case. For this experiment, we double the workload intensity halfway through the trace. The cluster is then scaled such that the system utilization after scaling is the same as the system utilization before scaling, 0.8 in our case.

Figure 12a depicts the performance of three architecture before and after scaling. We observe that there is no significant difference in task slowdown before and after scaling. Figure 12b depicts the additional resources required after scaling to maintain the same resource utilization. We observe that a median of 4% additional resource are required to maintain the same resource utilization, and hence obtain the same performance.

Key takeaway: We exemplify that scaling the number of nodes according to the workload as a valid approach with reasonable overhead to keep performance stable in a serverless data-processing context.

5.5. Impact of Object Popularity

Skewed object popularity is present in the real-world traces [12] and impacting the resulting performance. In Figure 13, we plot the median (left) and tail (right) slowdown compared to object popularity skew for the three architecture variants in comparison. Object popularity skew measures the fraction of tasks that access the top 1% objects. In other words, 0.1 means that 10% of accesses go to the top 1% objects. The x-axis is cut at 0.4 as none of our traces exhibit a higher skew, except one outlier trace.

The negative impact on performance for increasing skew value is visible both in median and tail for all three architecture variants. The centralized and delegated architectures both cope in a similar way with the skewed object accesses. With few exceptions but matching expert intuition, the decentralized architecture is suffering most from skewed object accesses both in median and tail slowdown.

Key takeaway: Centralized and delegated architectures can handle skewed object popularity better than the decentralized architecture, when not using additional mechanisms.

6. Validating the Simulation

To validate our simulations, we compare the results from one of our experiments with those from a real-world setup. Specifically, we compare the task slowdown at 0.8 utilization and 4 tasks per node for all three architectures. In the real-world setup, we deploy the scheduler in an OpenWhisk-like [29] serverless system. The system makes a network call to our scheduler for every scheduling decision. In the real-world setup, we mock the data processing and the latency to remote storage with the

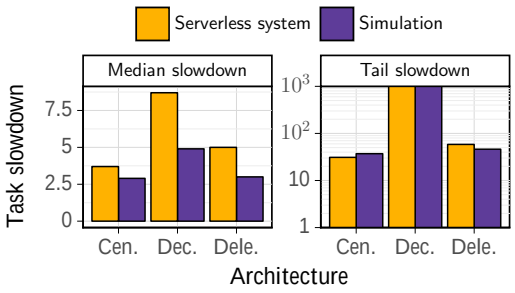


Figure 14: Comparing median and tail task slowdown between a serverless system implementation and the simulation, for one trace at 0.8 utilization and 4 tasks per node.

same parameters we use in the simulation (Section 4). Mocking the data processing allows us to run more workers than we have physical cores available. All scheduler components are deployed on the same physical machine: An Ubuntu 22.04 server with two Xeon 4210R CPUs and 256GB of RAM. We run 17 workers executing the tasks and 1 (centralized) to 5 (delegated and decentralized) scheduler nodes based on the experiment configuration.

Generally, we notice the real-world experiments have a much higher overhead per task than we modeled in the simulator. We believe these overheads in the real system can be reduced using busy polling, a thread per core architecture, and a better OS scheduler [16]. Nevertheless, the overheads we observe are representative of deployed systems that are not highly optimized. We plan to add models for these systems with higher overheads to the simulator. We also noticed queue build-ups when we receive task completion acknowledgment in the load generation application. Therefore, for now, we separate the acknowledgments from load generation. We plan to investigate this further.

Figure 14 compares the serverless system results with the simulation results for the same trace. In the right plot (logarithmic scale), we observe that the high tail slowdown of the decentralized architecture in simulation is also matched by the serverless system. The tail slowdown is 16% lower for the centralized architecture in the serverless system. On the other hand, the tail slowdown is 26% higher in the serverless system for the delegated architecture. Given the overheads in the serverless system, we consider results in the same order of magnitude and a maximum difference of 26% valid for the tail slowdown.

In the left plot, we observe that the centralized architecture experiences a 20% lower median slowdown in the serverless system. The decentralized architecture experiences a 78% higher median slowdown in the serverless system compared to simulation. The decentralized architecture experiences a 26% higher median slowdown. Despite the differences, we observe that the relative order of performance across the three architectures remains the same.

Based on the similar performance trend across three architectures we observe in both the serverless system and the simulation, we consider the results of our characterization indicative of real-world performance. A better simulation model that takes into account the considerable overhead of real systems is necessary before the precise performance measurements from the simulator can be used in a real-world setting.

7. Threats to Validity

Incomplete conceptual model: We demonstrated that the scheduler frame conceptual model covers architectures and six different mechanisms in Section 2. But, there is no proof that the conceptual model is complete and exhaustive. We define a frame as the set of mechanisms which cannot be implemented without communication between scheduler components. The communication constraint is only a necessary condition and not sufficient. Our model does not cover specific bookkeeping and implementation changes should be made to the components.
Simulator validity: We use traces and realistic parameters for our experiments. We also validate one of our experiments by comparing with the results of a real-world emulation. But, the results we obtain are only indicative of real-world performance. A better network model and a sensitivity study of the network-related parameters would improve the applicability and generality of our results.
Other scheduling policies: We characterize three scheduler architectures and two mechanisms. But, our characterization uses only the FIFO scheduling policy. Scheduling policies have a major impact on system performance [24]. Other, more sophisticated, scheduling policies can lead to increase or decrease in the performance gap between different architectures and can even result in an inversion in the performance ranking. Our framework can be used to evaluate different scheduling policies across multiple architectures quickly and consistently.

8. Related Work

Simulation: Closest to our work are existing simulators like CloudSim [45], GangSim [44], and OpenDC [22]. CloudSim has been particularly successful with its many extensions for fog computing [57], data placement [58], fault tolerance [59], energy awareness [60], and more. These extensions indicate that adding new architectures and mechanisms to simulators like CloudSim required significant effort. We propose the conceptual model of scheduler frames to reduce this effort. We enable easier scheduler mechanism design and exploration by providing first-class support for inter-component communication in our conceptual model and our simulator.
Systematic characterization: Ahmad and Kwok performed one of the early characterization of task graph scheduling algorithms [42]. But, they do not consider multiple architectures and mechanisms. Serverless scheduling [24] and multi-core scheduling [16] have been systematically characterized recently. Both studies use the centralized scheduler architecture. Both characterize task placement policies and resource allocation policies. We characterize multiple scheduler architectures, and the work stealing and data migration mechanisms. Work stealing proved to be a successful mechanism for multi-core scheduling [16], just like it did for serverless scheduling in this work.
Scheduler architectures: Many scheduler designs which use different architectures exist: centralized [28, 29], decentralized [30,

4, 61], delegated [31, 13, 10], and hybrid [32, 33, 62]. Each scheduler explores different trade-offs and their ideas were integrated into other schedulers that succeeded them. We propose a model to integrate the different architectures into one conceptual model to ease design space exploration without building the whole system. All the distributed scheduler papers explore the impact of distribution on fair resource allocation, which we do not explore in this work.

Serverless scheduling: Many existing serverless systems using centralized scheduling [29, 63, 64, 65, 24], but other designs have been proposed [31, 1]. Some serverless systems implement work stealing [66, 65], but only for the centralized architecture. We characterize the impact of work stealing across different architectures. Data management for serverless systems has seen much exploration [21, 67, 68, 69]. We only explore the data migration technique, and not others such as fusion and prefetching.

9. Conclusion

The scheduler is a crucial component in serverless data-processing systems: its architecture, mechanisms, and their interactions can dramatically impact the system's performance.

In this work, we proposed `ExDe`, a framework to help system designers explore the vast design space of scheduler architectures and mechanisms thoroughly and conveniently. Reasoning on the concept of scheduler frames, the users explicitly define and explore actions that require the coordination of multiple scheduler components to occur. `ExDe` relies on discrete-event simulation to evaluate different frames in isolation and conduct thousands of performance evaluations in a timely and cost-effective manner.

`ExDe` is open-source and thanks to its interface, can be easily extended by practitioners, favoring the explorations of other different variants of schedulers. It is available at `https://zenodo.org/record/7829151`.

Acknowledgements

This research was partially supported by the EU MSCA Cloudstars grant. This research was partially supported by the AWS Cloud Credit for Research program.

References

[1] R. Chard, Y. N. Babuji, Z. Li, T. J. Skluzacek, A. Woodard, B. Blaiszik, I. T. Foster, K. Chard, funcx: A federated function serving fabric for science, in: M. Parashar, V. Vlassov, D. E. Irwin, K. Mohror (Eds.), HPDC '20: The 29th International Symposium on High-Performance Parallel and Distributed Computing, Stockholm, Sweden, June 23-26, 2020, ACM, 2020, pp. 65–76. doi:10.1145/3369583.3392683.
URL `https://doi.org/10.1145/3369583.3392683`

[2] I. Müller, R. Marroquín, G. Alonso, Lambada: Interactive data analytics on cold data using serverless cloud infrastructure, in: D. Maier, R. Pottinger, A. Doan, W. Tan, A. Alawini, H. Q. Ngo (Eds.), Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020, ACM, 2020, pp. 115–130. doi:10.1145/3318464.3389758.
URL `https://doi.org/10.1145/3318464.3389758`

[3] K. Rodrigues, Y. Luo, D. Yuan, CLP: efficient and scalable search on compressed text logs, in: A. D. Brown, J. R. Lorch (Eds.), 15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021, USENIX Association, 2021, pp. 183–198.
URL `https://www.usenix.org/conference/osdi21/presentation/rodrigues`

[4] M. Vuppalapati, J. Miron, R. Agarwal, D. Truong, A. Motivala, T. Cruanes, Building an elastic query engine on disaggregated storage, in: R. Bhagwan, G. Porter (Eds.), 17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020, USENIX Association, 2020, pp. 449–462.
URL `https://www.usenix.org/conference/nsdi20/presentation/vuppalapati`

[5] E. Jonas, J. Schleier-Smith, V. Sreekanti, C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. J. Yadwadkar, J. E. Gonzalez, R. A. Popa, I. Stoica, D. A. Patterson, Cloud programming simplified: A berkeley view on serverless computing, CoRR abs/1902.03383 (2019). arXiv:1902.03383.
URL `http://arxiv.org/abs/1902.03383`

[6] Serverless: What it is, `https://glossary.cncf.io/serverless/`, accessed: 10-10-2022 (2022).

[7] E. V. Eyk, L. Toader, S. Talluri, L. Versluis, A. Uta, A. Iosup, Serverless is more: From paas to present cloud computing, IEEE Internet Comput. 22 (5) (2018) 8–17. doi:10.1109/MIC.2018.053681358.
URL `https://doi.org/10.1109/MIC.2018.053681358`

[8] M. Perron, R. C. Fernandez, D. J. DeWitt, S. Madden, Starling: A scalable query engine on cloud functions, in: D. Maier, R. Pottinger, A. Doan, W. Tan, A. Alawini, H. Q. Ngo (Eds.), Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020, ACM, 2020, pp. 131–141. doi:10.1145/3318464.3380609.
URL `https://doi.org/10.1145/3318464.3380609`

[9] Databricks serverless compute, `https://docs.databricks.com/serverless-compute/index.html`, accessed: 10-10-2022 (2022).

[10] A. Iosup, D. H. J. Epema, T. Tannenbaum, M. Farrellee, M. Livny, Inter-operating grids through delegated matchmaking, in: B. Verastegui (Ed.), Proceedings of the ACM/IEEE Conference on High Performance Networking and Computing, SC 2007, November 10-16, 2007, Reno, Nevada, USA, ACM Press, 2007, p. 13. doi:10.1145/1362622.1362640.
URL `https://doi.org/10.1145/1362622.1362640`

[11] R. Levin, E. S. Cohen, W. M. Corwin, F. J. Pollack, W. A. Wulf, Policy/mechanism separation in HYDRA, in: J. C. Browne, J. Rodriguez-Rosell (Eds.), Proceedings of the Fifth Symposium on Operating System Principles, SOSP 1975, The University of Texas at Austin, Austin, Texas, USA, November 19-21, 1975, ACM, 1975, pp. 132–140. doi:10.1145/800213.806531.
URL `https://doi.org/10.1145/800213.806531`

[12] O. Eytan, D. Harnik, E. Ofer, R. Friedman, R. I. Kat, It's time to revisit LRU vs. FIFO, in: A. Badam, V. Chidambaram (Eds.), 12th USENIX Workshop on Hot Topics in Storage and File Systems, HotStorage 2020, July 13-14, 2020, USENIX Association, 2020.
URL `https://www.usenix.org/conference/hotstorage20/presentation/eytan`

[13] D. Thain, T. Tannenbaum, M. Livny, Distributed computing in practice: the condor experience, Concurr. Pract. Exp. 17 (2-4) (2005) 323–356. doi:10.1002/cpe.938.
URL `https://doi.org/10.1002/cpe.938`

[14] B. Burns, B. Grant, D. Oppenheimer, E. A. Brewer, J. Wilkes, Borg, omega, and kubernetes, Commun. ACM 59 (5) (2016) 50–57. doi:10.1145/2890784.
URL `https://doi.org/10.1145/2890784`

[15] G. Andreadis, L. Versluis, F. Mastenbroek, A. Iosup, A reference architecture for datacenter scheduling: design, validation, and experiments, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, SC 2018, Dallas, TX, USA, November 11-16, 2018, IEEE / ACM, 2018, pp. 37:1–37:15.
URL `http://dl.acm.org/citation.cfm?id=3291706`

[16] S. McClure, A. Ousterhout, S. Shenker, S. Ratnasamy, Efficient scheduling policies for microsecond-scale tasks, in: A. Phanishayee, V. Sekar (Eds.), 19th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2022, Renton, WA, USA, April 4-6, 2022,

USENIX Association, 2022, pp. 1–18.
URL https://www.usenix.org/conference/nsdi22/presentation/mcclure

[17] J. M. Schopf, A general architecture for scheduling on the grid, Special issue of JPDC on Grid Computing 4 (2002).

[18] R. Bhardwaj, A. Tumanov, S. Wang, R. Liaw, P. Moritz, R. Nishihara, I. Stoica, ESCHER: expressive scheduling with ephemeral resources, in: A. Gavrilovska, D. Altinbüken, C. Binnig (Eds.), Proceedings of the 13th Symposium on Cloud Computing, SoCC 2022, San Francisco, California, November 7-11, 2022, ACM, 2022, pp. 47–62. doi:10.1145/3542929.3563498.
URL https://doi.org/10.1145/3542929.3563498

[19] A. M. Lasa, S. Talluri, A. Iosup, A reference architecture for datacenter scheduler programming abstractions: Design and experiments (work in progress paper), in: ACM/SPEC International Conference on Performance Engineering 2023, ACM, 2023.

[20] V. M. Gracia, O. F. Rana, J. Á. Bañares, U. Arronategui, Modelling performance & resource management in kubernetes, in: C. Jiang, O. F. Rana, N. Antonopoulos (Eds.), Proceedings of the 9th International Conference on Utility and Cloud Computing, UCC 2016, Shanghai, China, December 6-9, 2016, ACM, 2016, pp. 257–262. doi:10.1145/2996890.3007869.
URL https://doi.org/10.1145/2996890.3007869

[21] A. Klimovic, Y. Wang, P. Stuedi, A. Trivedi, J. Pfefferle, C. Kozyrakis, Pocket: Elastic ephemeral storage for serverless analytics, in: A. C. Arpaci-Dusseau, G. Voelker (Eds.), 13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018, USENIX Association, 2018, pp. 427–444.
URL https://www.usenix.org/conference/osdi18/presentation/klimovic

[22] F. Mastenbroek, G. Andreadis, S. Jounaid, W. Lai, J. Burley, J. Bosch, E. V. Eyk, L. Versluis, V. van Beek, A. Iosup, Opendc 2.0: Convenient modeling and simulation of emerging technologies in cloud datacenters, in: L. Lefèvre, S. Patterson, Y. C. Lee, H. Shen, S. Ilager, M. Goudarzi, A. N. Toosi, R. Buyya (Eds.), 21st IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing, CCGrid 2021, Melbourne, Australia, May 10-13, 2021, IEEE, 2021, pp. 455–464. doi:10.1109/CCGrid51090.2021.00055.
URL https://doi.org/10.1109/CCGrid51090.2021.00055

[23] A. Mahgoub, E. B. Yi, K. Shankar, E. Minocha, S. Elnikety, S. Bagchi, S. Chaterji, WISEFUSE: workload characterization and DAG transformation for serverless workflows, Proc. ACM Meas. Anal. Comput. Syst. 6 (2) (2022) 26:1–26:28. doi:10.1145/3530892.
URL https://doi.org/10.1145/3530892

[24] K. Kaffes, N. J. Yadwadkar, C. Kozyrakis, Hermod: principled and practical scheduling for serverless functions, in: A. Gavrilovska, D. Altinbüken, C. Binnig (Eds.), Proceedings of the 13th Symposium on Cloud Computing, SoCC 2022, San Francisco, California, November 7-11, 2022, ACM, 2022, pp. 289–305. doi:10.1145/3542929.3563468.
URL https://doi.org/10.1145/3542929.3563468

[25] H. Bian, A. Ailamaki, Pixels: An efficient column store for cloud data lakes, in: 38th IEEE International Conference on Data Engineering, ICDE 2022, Kuala Lumpur, Malaysia, May 9-12, 2022, IEEE, 2022, pp. 3078–3090. doi:10.1109/ICDE53745.2022.00276.
URL https://doi.org/10.1109/ICDE53745.2022.00276

[26] R. B. Roy, T. Patel, D. Tiwari, Characterizing and mitigating the I/O scalability challenges for serverless applications, in: IEEE International Symposium on Workload Characterization, IISWC 2021, Storrs, CT, USA, November 7-9, 2021, IEEE, 2021, pp. 74–86. doi:10.1109/IISWC53511.2021.00018.
URL https://doi.org/10.1109/IISWC53511.2021.00018

[27] Q. Pu, S. Venkataraman, I. Stoica, Shuffling, fast and slow: Scalable analytics on serverless infrastructure, in: J. R. Lorch, M. Yu (Eds.), 16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019, USENIX Association, 2019, pp. 193–206.
URL https://www.usenix.org/conference/nsdi19/presentation/pu

[28] D. B. Jackson, Q. Snell, M. J. Clement, Core algorithms of the maui scheduler, in: D. G. Feitelson, L. Rudolph (Eds.), Job Scheduling Strategies for Parallel Processing, 7th International Workshop, JSSPP 2001, Cambridge, MA, USA, June 16, 2001, Revised Papers, Vol. 2221 of Lecture Notes in Computer Science, Springer, 2001, pp. 87–102. doi:10.1007/3-540-45540-X_6.
URL https://doi.org/10.1007/3-540-45540-X_6

[29] Apache whisk: Open source serverless cloud platform, https://openwhisk.apache.org/, accessed: 10-10-2022 (2022).

[30] A. Fuerst, P. Sharma, Locality-aware load-balancing for serverless clusters, in: J. B. Weissman, A. Chandra, A. Gavrilovska, D. Tiwari (Eds.), HPDC '22: The 31st International Symposium on High-Performance Parallel and Distributed Computing, Minneapolis, MN, USA, 27 June 2022 - 1 July 2022, ACM, 2022, pp. 227–239. doi:10.1145/3502181.3531459.
URL https://doi.org/10.1145/3502181.3531459

[31] A. Singhvi, A. Balasubramanian, K. Houck, M. D. Shaikh, S. Venkataraman, A. Akella, Atoll: A scalable low-latency serverless platform, in: C. Curino, G. Koutrika, R. Netravali (Eds.), SoCC '21: ACM Symposium on Cloud Computing, Seattle, WA, USA, November 1 - 4, 2021, ACM, 2021, pp. 138–152. doi:10.1145/3472883.3486981.
URL https://doi.org/10.1145/3472883.3486981

[32] V. A. Olteanu, A. Agache, A. Voinescu, C. Raiciu, Stateless datacenter load-balancing with beamer, in: S. Banerjee, S. Seshan (Eds.), 15th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2018, Renton, WA, USA, April 9-11, 2018, USENIX Association, 2018, pp. 125–139.
URL https://www.usenix.org/conference/nsdi18/presentation/olteanu

[33] P. Delgado, F. Dinu, A. Kermarrec, W. Zwaenepoel, Hawk: Hybrid datacenter scheduling, in: S. Lu, E. Riedel (Eds.), 2015 USENIX Annual Technical Conference, USENIX ATC '15, July 8-10, Santa Clara, CA, USA, USENIX Association, 2015, pp. 499–510.
URL https://www.usenix.org/conference/atc15/technical-session/presentation/delgado

[34] H. Topcuoglu, S. Hariri, M. Wu, Performance-effective and low-complexity task scheduling for heterogeneous computing, IEEE Trans. Parallel Distributed Syst. 13 (3) (2002) 260–274. doi:10.1109/71.993206.
URL https://doi.org/10.1109/71.993206

[35] S. Viswanathan, B. Veeravalli, T. G. Robertazzi, Resource-aware distributed scheduling strategies for large-scale computational cluster/grid systems, IEEE Trans. Parallel Distributed Syst. 18 (10) (2007) 1450–1461. doi:10.1109/TPDS.2007.1073.
URL https://doi.org/10.1109/TPDS.2007.1073

[36] J. Stojkovic, T. Xu, H. Franke, J. Torrellas, Specfaas: Accelerating serverless applications with speculative function execution, in: IEEE International Symposium on High-Performance Computer Architecture, HPCA 2023, Montreal, QC, Canada, February 25 - March 1, 2023, IEEE, 2023, pp. 814–827. doi:10.1109/HPCA56546.2023.10071120.
URL https://doi.org/10.1109/HPCA56546.2023.10071120

[37] S. Burckhardt, B. Chandramouli, C. Gillum, D. Justo, K. Kallas, C. McMahon, C. Meiklejohn, X. Zhu, Netherite: Efficient execution of serverless workflows, Proc. VLDB Endow. 15 (8) (2022) 1591–1604.
URL https://www.vldb.org/pvldb/vol15/p1591-burckhardt.pdf

[38] V. Sreekanti, C. Wu, S. Chhatrapati, J. E. Gonzalez, J. M. Hellerstein, J. M. Faleiro, A fault-tolerance shim for serverless computing, in: A. Bilas, K. Magoutis, E. P. Markatos, D. Kostic, M. I. Seltzer (Eds.), EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020, ACM, 2020, pp. 15:1–15:15. doi:10.1145/3342195.3387535.
URL https://doi.org/10.1145/3342195.3387535

[39] J. Chen, H. Zhang, W. Zhang, L. Luo, J. S. Chase, I. Stoica, D. Zhuo, Nethint: White-box networking for multi-tenant data centers, in: A. Phanishayee, V. Sekar (Eds.), 19th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2022, Renton, WA, USA, April 4-6, 2022, USENIX Association, 2022, pp. 1327–1343.
URL https://www.usenix.org/conference/nsdi22/presentation/chen-jingrong

[40] D. G. Feitelson, L. Rudolph, Gang scheduling performance benefits for fine-grain synchronization, J. Parallel Distributed Comput. 16 (4) (1992) 306–318. doi:10.1016/0743-7315(92)90014-E.
URL https://doi.org/10.1016/0743-7315(92)90014-E

[41] I. Gog, M. Schwarzkopf, A. Gleave, R. N. M. Watson, S. Hand, Firmament: Fast, centralized cluster scheduling at scale, in: K. Keeton, T. Roscoe (Eds.), 12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November

2-4, 2016, USENIX Association, 2016, pp. 99–115.
URL https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gog

[42] Y. Kwok, I. Ahmad, Benchmarking and comparison of the task graph scheduling algorithms, J. Parallel Distributed Comput. 59 (3) (1999) 381–422. doi:10.1006/jpdc.1999.1578.
URL https://doi.org/10.1006/jpdc.1999.1578

[43] Z. Zhan, X. F. Liu, Y. Gong, J. Zhang, H. S. Chung, Y. Li, Cloud computing resource scheduling and a survey of its evolutionary approaches, ACM Comput. Surv. 47 (4) (2015) 63:1–63:33. doi:10.1145/2788397.
URL https://doi.org/10.1145/2788397

[44] C. Dumitrescu, I. T. Foster, Gangsim: a simulator for grid scheduling studies, in: 5th International Symposium on Cluster Computing and the Grid (CCGrid 2005), 9-12 May, 2005, Cardiff, UK, IEEE Computer Society, 2005, pp. 1151–1158. doi:10.1109/CCGRID.2005.1558689.
URL https://doi.org/10.1109/CCGRID.2005.1558689

[45] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. D. Rose, R. Buyya, Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms, Softw. Pract. Exp. 41 (1) (2011) 23–50. doi:10.1002/spe.995.
URL https://doi.org/10.1002/spe.995

[46] Y. Artsy, M. Livny, An approach to the design of fully open computing systems, Tech. rep., University of Wisconsin-Madison Department of Computer Sciences (1987).

[47] M. Abdi, S. Ginzburg, X. C. Lin, J. M. Faleiro, G. I. Chaudhry, I. Goiri, R. Bianchini, D. S. Berger, R. Fonseca, Palette load balancing: Locality hints for serverless functions, in: G. A. D. Luna, L. Querzoni, A. Fedorova, D. Narayanan (Eds.), Proceedings of the Eighteenth European Conference on Computer Systems, EuroSys 2023, Rome, Italy, May 8-12, 2023, ACM, 2023, pp. 365–380. doi:10.1145/3552326.3567496.
URL https://doi.org/10.1145/3552326.3567496

[48] X. Sun, W. Luo, J. T. Gu, A. Ganesan, R. Alagappan, M. Gasch, L. Suresh, T. Xu, Automatic reliability testing for cluster management controllers, in: M. K. Aguilera, H. Weatherspoon (Eds.), 16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022, USENIX Association, 2022, pp. 143–159.
URL https://www.usenix.org/conference/osdi22/presentation/sun

[49] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, J. Wilkes, Omega: flexible, scalable schedulers for large compute clusters, in: Z. Hanzálek, H. Härtig, M. Castro, M. F. Kaashoek (Eds.), Eighth Eurosys Conference 2013, EuroSys '13, Prague, Czech Republic, April 14-17, 2013, ACM, 2013, pp. 351–364. doi:10.1145/2465351.2465386.
URL https://doi.org/10.1145/2465351.2465386

[50] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, I. Stoica, Ray: A distributed framework for emerging AI applications, in: A. C. Arpaci-Dusseau, G. Voelker (Eds.), 13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018, USENIX Association, 2018, pp. 561–577.
URL https://www.usenix.org/conference/osdi18/presentation/nishihara

[51] W. McKinney, et al., pandas: a foundational python library for data analysis and statistics, Python for high performance and scientific computing 14 (9) (2011) 1–9.

[52] D. D. Sensi, T. D. Matteis, K. Taranov, S. D. Girolamo, T. Rahn, T. Hoefler, Noise in the clouds: Influence of network performance variability on application scalability, Proc. ACM Meas. Anal. Comput. Syst. 6 (3) (2022) 49:1–49:27. doi:10.1145/3570609.
URL https://doi.org/10.1145/3570609

[53] S. Snyder, P. H. Carns, R. Latham, M. Mubarak, R. B. Ross, C. D. Carothers, B. Behzad, H. V. T. Luu, S. Byna, Prabhat, Techniques for modeling large-scale HPC I/O workloads, in: S. A. Jarvis, S. A. Wright, S. D. Hammond (Eds.), Proceedings of the 6th International Workshop on Performance Modeling, Benchmarking, and Simulation of High Performance Computing Systems, PMBS 2015, Austin, Texas, USA, November 15, 2015, ACM, 2015, pp. 5:1–5:11. doi:10.1145/2832087.2832091.
URL https://doi.org/10.1145/2832087.2832091

[54] R. B. Roy, T. Patel, D. Tiwari, Daydream: Executing dynamic scientific workflows on serverless platforms with hot starts, in: SC22: International Conference for High Performance Computing, Networking, Storage and Analysis, Dallas, TX, USA, November 13-18, 2022, IEEE, 2022, pp. 1–18. doi:10.1109/SC41404.2022.00027.
URL https://doi.org/10.1109/SC41404.2022.00027

[55] A. Uta, A. Custura, D. Duplyakin, I. Jimenez, J. S. Rellermeyer, C. Maltzahn, R. Ricci, A. Iosup, Is big data performance reproducible in modern cloud networks?, in: R. Bhagwan, G. Porter (Eds.), 17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020, USENIX Association, 2020, pp. 513–527.
URL https://www.usenix.org/conference/nsdi20/presentation/uta

[56] E. Frachtenberg, D. G. Feitelson, Pitfalls in parallel job scheduling evaluation, in: D. G. Feitelson, E. Frachtenberg, L. Rudolph, U. Schwiegelshohn (Eds.), Job Scheduling Strategies for Parallel Processing, 11th International Workshop, JSSPP 2005, Cambridge, MA, USA, June 19, 2005, Revised Selected Papers, Vol. 3834 of Lecture Notes in Computer Science, Springer, 2005, pp. 257–282. doi:10.1007/11605300_13.
URL https://doi.org/10.1007/11605300_13

[57] M. R. Mahmud, S. Pallewatta, M. Goudarzi, R. Buyya, ifogsim2: An extended ifogsim simulator for mobility, clustering, and microservice management in edge and fog computing environments, J. Syst. Softw. 190 (2022) 111351. doi:10.1016/j.jss.2022.111351.
URL https://doi.org/10.1016/j.jss.2022.111351

[58] M. I. Naas, J. Boukhobza, P. R. Parvédy, L. Lemarchand, An extension to ifogsim to enable the design of data placement strategies, in: 2nd IEEE International Conference on Fog and Edge Computing, ICFEC 2018, Washington DC, USA, May 1-3, 2018, IEEE, 2018, pp. 1–8. doi:10.1109/CFEC.2018.8358724.
URL https://doi.org/10.1109/CFEC.2018.8358724

[59] M. Jammal, H. Hawilo, A. Kanso, A. Shami, ACE: availability-aware cloudsim extension, IEEE Trans. Netw. Serv. Manag. 15 (4) (2018) 1586–1599. doi:10.1109/TNSM.2018.2879665.
URL https://doi.org/10.1109/TNSM.2018.2879665

[60] D. Kliazovich, P. Bouvry, S. U. Khan, Greencloud: a packet-level simulator of energy-aware cloud computing data centers, J. Supercomput. 62 (3) (2012) 1263–1283. doi:10.1007/s11227-010-0504-1.
URL https://doi.org/10.1007/s11227-010-0504-1

[61] E. Boutin, J. Ekanayake, W. Lin, B. Shi, J. Zhou, Z. Qian, M. Wu, L. Zhou, Apollo: Scalable and coordinated scheduling for cloud-scale computing, in: J. Flinn, H. Levy (Eds.), 11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014, USENIX Association, 2014, pp. 285–300.
URL https://www.usenix.org/conference/osdi14/technical-sessions/presentation/boutin

[62] K. Karanasos, S. Rao, C. Curino, C. Douglas, K. Chaliparambil, G. M. Fumarola, S. Heddaya, R. Ramakrishnan, S. Sakalanaga, Mercury: Hybrid centralized and distributed scheduling in large shared clusters, in: S. Lu, E. Riedel (Eds.), 2015 USENIX Annual Technical Conference, USENIX ATC '15, July 8-10, Santa Clara, CA, USA, USENIX Association, 2015, pp. 485–497.
URL https://www.usenix.org/conference/atc15/technical-session/presentation/karanasos

[63] B. Carver, J. Zhang, A. Wang, A. Anwar, P. Wu, Y. Cheng, Wukong: a scalable and locality-enhanced framework for serverless parallel computing, in: R. Fonseca, C. Delimitrou, B. C. Ooi (Eds.), SoCC '20: ACM Symposium on Cloud Computing, Virtual Event, USA, October 19-21, 2020, ACM, 2020, pp. 1–15. doi:10.1145/3419111.3421286.
URL https://doi.org/10.1145/3419111.3421286

[64] A. Tariq, A. Pahl, S. Nimmagadda, E. Rozner, S. Lanka, Sequoia: enabling quality-of-service in serverless computing, in: R. Fonseca, C. Delimitrou, B. C. Ooi (Eds.), SoCC '20: ACM Symposium on Cloud Computing, Virtual Event, USA, October 19-21, 2020, ACM, 2020, pp. 311–327. doi:10.1145/3419111.3421306.
URL https://doi.org/10.1145/3419111.3421306

[65] K. Kaffes, N. J. Yadwadkar, C. Kozyrakis, Centralized core-granular scheduling for serverless functions, in: Proceedings of the ACM Symposium on Cloud Computing, SoCC 2019, Santa Cruz, CA, USA, November 20-23, 2019, ACM, 2019, pp. 158–164. doi:10.1145/3357223.3362709.

URL https://doi.org/10.1145/3357223.3362709

[66] P. K. Gadepalli, S. McBride, G. Peach, L. Cherkasova, G. Parmer, Sledge: a serverless-first, light-weight wasm runtime for the edge, in: D. D. Silva, R. Kapitza (Eds.), Middleware '20: 21st International Middleware Conference, Delft, The Netherlands, December 7-11, 2020, ACM, 2020, pp. 265–279. doi:10.1145/3423211.3425680.
URL https://doi.org/10.1145/3423211.3425680

[67] A. Mahgoub, K. Shankar, S. Mitra, A. Klimovic, S. Chaterji, S. Bagchi, SONIC: application-aware data passing for chained serverless applications, in: I. Calciu, G. Kuenning (Eds.), 2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021, USENIX Association, 2021, pp. 285–301.
URL https://www.usenix.org/conference/atc21/presentation/mahgoub

[68] D. Mvondo, M. Bacou, K. Nguetchouang, L. Ngale, S. Pouget, J. Kouam, R. Lachaize, J. Hwang, T. Wood, D. Hagimont, N. D. Palma, B. Batchakui, A. Tchana, OFC: an opportunistic caching system for faas platforms, in: A. Barbalace, P. Bhatotia, L. Alvisi, C. Cadar (Eds.), EuroSys '21: Sixteenth European Conference on Computer Systems, Online Event, United Kingdom, April 26-28, 2021, ACM, 2021, pp. 228–244. doi:10.1145/3447786.3456239.
URL https://doi.org/10.1145/3447786.3456239

[69] F. Romero, G. I. Chaudhry, I. Goiri, P. Gopa, P. Batum, N. J. Yadwadkar, R. Fonseca, C. Kozyrakis, R. Bianchini, Faa$t: A transparent auto-scaling cache for serverless applications, in: C. Curino, G. Koutrika, R. Netravali (Eds.), SoCC '21: ACM Symposium on Cloud Computing, Seattle, WA, USA, November 1 - 4, 2021, ACM, 2021, pp. 122–137. doi:10.1145/3472883.3486974.
URL https://doi.org/10.1145/3472883.3486974