ScalaSSC: Scalable Stateful Serverless Computing for Stream Processing Applications

Tianyu Qi, Maria A. Rodriguez, and Rajkumar Buyya
School of Computing and Information Systems
The University of Melbourne, Australia
tiqi@student.unimelb.edu.au, marodriguez@unimelb.edu.au, rbuyya@unimelb.edu.au

***Abstract*—Serverless platforms are increasingly being used to process continuous streams of data. However, data processing units (expressed as serverless 'functions') in such platforms cannot maintain state internally and, instead, rely on remote storage. Stateful serverless is an emerging paradigm that seeks to reduce the latency associated with remote storage access by introducing state servers on worker nodes, i.e., where the serverless functions run. While existing stateful serverless frameworks are successful in reducing latency compared to their stateless counterparts, there are multiple challenges that still need to be addressed in order to meet the real-time data processing requirements of streaming applications. As a result, this paper proposes ScalaSSC, a novel framework for data processing request scheduling and operator state management tailored to stream processing applications within stateful serverless computing. ScalaSSC co-locates requests with the state they act upon to avoid cross-worker state access. To reduce state access contention among executors, ScalaSSC not only introduces the concept of state parallelism to serverless computing, but also processes requests acting upon the same state in the same batch. During batch execution, when multiple states are required, batch APIs are provided to access all states in a single operation. Experimental results show that ScalaSSC achieves up to 845 times higher throughput than another stateful serverless system while maintaining similar end-to-end latency. It also achieves a throughput 14 times higher than that of Amazon Lambda while maintaining end-to-end latency at the microsecond level, in contrast to Lambda's latency at the second level.**

***Index Terms*—Serverless Computing, FaaS, Stateful Serverless, Big Data, Stream Processing.**

I. Introduction

With the advent of the big data era, the volume of data generated has exponentially increased over the past decade [17]. These data sets contain potentially valuable information that is not known in advance and requires analysis to extract insights. Consequently, big data technologies, such as batch and stream processing, have become essential for organizations and researchers aiming to uncover valuable knowledge from data [15]. Examples of such applications include identifying Twitter trends [21], traffic monitoring [32], and pandemic control [7].

To effectively handle and analyze continuous data streams in real-time, Distributed Stream Processing Systems (DSPSs) have been developed with the goals of achieving low latency and high throughput. In DSPSs, operators, which are application-level entities that encapsulate data processing logic, are deployed in long-running executors. These executors are considered to be heavy-weight as they maintain system-level state (e.g., inbound and outbound data streams from/to other executors) and application-level state specific to an operator's logic. Users are required to manage the infrastructure and schedule executors, with the latter being an NP-hard task [11]. Scaling the number of executors (i.e., operator instances) out enhances the processing capacity of an application deployed in a DSPS, thus allowing it to effectively manage higher input rates. The reason for this is two-fold. First, scaling results in increased data processing components available in the system. Second, and most importantly, the data streams and application's state are distributed and independently managed across operator instances, thus minimising contention and maximising concurrency. However, creating, removing, or migrating these heavy-weight executors incurs substantial overhead, limiting the ability of applications to adapt to fluctuating input rates [28].

By addressing the aforementioned limitations of DSPSs, serverless computing (through its function-as-a-service offering) has seen widespread adoption in the stream processing domain, including in fields such as data analytics [3], [20], machine learning [29], and IoT sensor monitoring [10], [13], [34]. In a serverless environment, data processing operators execute within *functions*. Functions are considered lightweight and ephemeral as they disaggregate storage and computing [14] (i.e., are stateless) and incur minimal deployment overhead. These features enable the rapid auto-scaling of operator instances in response to workload fluctuations and allow for instances to be flexibly distributed across nodes to meet resource demands. However, they also require stateful streaming applications to rely on remote storage, which ultimately has a negative effect on latency and throughput.

Stateful serverless is an emerging paradigm that aims to provide functions with system-managed storage within the worker nodes [19], [25], [27]. While it generally decreases state access latency compared to the stateless paradigm, there remain challenges that need to be addressed in order to optimize its performance for streaming applications. First, the majority of existing stateful approaches replicate state across multiple workers to alleviate the high latency associated with cross-worker state access. However, streaming applications update their state frequently, commonly after processing each input (i.e., data tuple), thus leading to significant overhead associated with maintaining consistency across state replicas. Second, existing stateful serverless systems focus solely on scaling operator instances when more processing capacity is required. The performance gains observed from increasing processing power are limited due to state access contention. Finally, processing multiple tuples in batch can reduce the workload associated with state loading [1], [35]. In the streaming application, certain tuple acts upon only a subset of the operator state. However, existing approaches either lack support for batching or simply batch tuples based on arrival time, which may result in increased state access operations and state access contention among executors.

To address these challenges, we propose ScalaSSC[1], a framework designed for stream processing applications in stateful serverless systems, with the goal of achieving low latency and high throughput. We classify application operators into three types: stateless, stateful, and partitioned stateful [12], and apply tailored scheduling, batching, and execution strategies for the tuple to each type. When processing a tuple acts upon the state of previous intermediate results, we schedule incoming tuples to the worker that holds the relevant state, in order to avoid cross-worker access. To reduce state access contention under high input rate, we scale the operator state to multiple independently managed instances, each being acted upon by a sub-stream of tuples. Once dispatched to a worker, tuples acting upon the same state are aggregated and executed in the same batch. In batch execution, our proposed batch APIs enable access to multiple states in a single operation, reducing the communication overhead. To reduce state access contention among executors, an operator's state is divided into fine-grained partitions. This ensures that a given tuple does not contend for the state it does not act upon. We build ScalaSSC on the stateful serverless platform FAASM [25] and evaluate it using DSPBench [6] applications on FAASM and AWS Lambda, focusing on performance metrics like end-to-end latency and throughput. Evaluation results show a maximum throughput improvement of 845 times and 8 times over FAASM and Lambda, respectively, along with a reduction in end-to-end latency of up to 87% and 95% compared to both systems.

II. Related Work

Extensive research has been conducted on stateful serverless computing by integrating storage servers inside workers. Shillaker and Pietzuch [25] propose FAASM which distributes a memory storage server on each host. FAASM features a local memory tier accessible by functions in the same host and a global memory tier shared across all functions to store state. The consistency among multiple replicas of the same state is maintained by the master copy. Sreekanti et al. [27] maintain a local cache on every execution VM and guarantee consistency by leveraging both application workflow and operation timestamp. Mvondo et al. [19] introduce OFC, a RAM-based durable key-value store on each node, comprising a master and backup copy for each data item, whose consistency is maintained by using the version number. All of these systems maintain multiple copies of the same state. In streaming applications, the frequently updated state introduces significant costs for maintaining consistency between these copies. To reduce the significant access latency associated with remote storage, some research focused on designing memory storage serving as middleware between the serverless platform and remote storage [4], [5], [16]. These frameworks can be deployed on top of any serverless platforms and their functions bind data to the memory storage instead of the remote storage. However, they result in higher access latency compared to the integrated storage frameworks due to the network delay. FaasFlow [18] and HashCache [30] utilize memory storage to transfer data between functions and do not address concurrent access issues.

Considerable research has been conducted on batching in serverless computing for machine learning. Certain studies conduct batch execution of multiple requests within a single function to achieve higher throughput [1], [2], [33], [35]. By utilizing multiple processes, some research further batch executes multiple requests concurrently in one function, as described in [8], [31]. However, in streaming applications, the states that requests act upon can differ and are locked while processing. Both simply executing all requests to the same operator in one batch and executing them concurrently cause unnecessary lock contention.

Several research studies have tailored serverless frameworks specifically to stream processing applications. Cai et al. [9] executes streaming applications on top of Lambda, using AWS DynamoDB as state storage. Song et al. [26] harnesses both virtual machines and serverless functions for streaming applications to meet latency requirements at a low monetary cost. However, both of these approaches lack adequate state management and access strategies to reduce access latency and state access contention between executors.

III. Background

This section provides background on streaming applications and defines the state models within ScalaSSC.

A. Operator Classification

We categorize streaming application operators into three types: stateless, stateful, and partitioned stateful.

A stateless operator processes each input tuple independently, without referencing previous processing results. In contrast, a stateful operator maintains an internal state to store intermediate results, referencing and updating this state as it processes new tuples. The operator state consists of multiple elements; for example, in Fig. 2, the *alert trigger* operator in the machine outlier application determines whether the input score indicates an anomaly based on past observations. Its state includes multiple elements: the previous timestamp, the minimum and maximum recorded scores, and a list of recorded observations.

A partitioned stateful operator is a commonly used specialized stateful operator. Bordin et al. [6] introduced a benchmark

[1]ScalaSSC is open-source and available at https://github.com/ScalaSSC/faasm

suite comprising 15 applications from various domains, 13 of which involve partitioned stateful operators. In a partitioned stateful operator, an attribute from the input tuple is defined as the partitioning attribute. Meanwhile, each element within the operator state is associated with a specific partitioning attribute value [12]. The operator logic ensures that the processing of each tuple acts upon only the state element associated with its partitioning attribute. For example, in the word count application, the partitioning attribute for the *counter* operator is the word itself, and each state element tracks the occurrence count for each word independently.

B. State Modeling

To execute stream processing applications in serverless systems, input data tuples are either encapsulated individually or grouped with other tuples targeting the same operator within an invocation request. In ScalaSSC, an input request R contains the invoked operator O and input tuples T, which comprises attribute A-value V pairs, represented as:

$$R = \{o, t_1, t_2, \ldots, t_n\}, \quad T = (a_1, v_1), (a_2, v_2), \ldots, (a_n, v_n)$$

For example, consider a request to the *speed calculator* operator which calculates the average car speed of each road in the traffic monitoring application [6]. The invoked operator o is *speed calculator*, with the first tuple t_1 containing attribute-value pairs where $a_1 = \texttt{road\_id}$ with v_1 represents the specific road ID, and $a_2 = \texttt{speed}$ with v_2 representing the speedometer value recorded by a car.

In our framework, we define the **operator state**, θ, to represent the set of elements E maintained by an operator. For a stateful operator o with n elements, the operator state is represented as:

$$\theta_o = \{e_1, e_2, e_3, \ldots, e_n\}$$

In DSPSs, parallelism refers to the number of instances of an operator, with each instance maintaining its own state partition of the operator state. In our framework, parallelism is defined as the number of operator state partitions. For the stateful operator with parallelism m, DSPSs and our approach replicate each element e_i into m replicas. In this context, the k^{th} operator state partition θ_o^k for operator o can be represented as follows:

$$\theta_o^k = \{e_1^k, e_2^k, e_3^k, \ldots, e_n^k\} \quad \forall k \in \{1, 2, \ldots, m\}$$

For the partitioned stateful operator, a partitioning attribute A_p is predefined. In both DSPSs and our approach, the state θ of a partitioned operator o includes an element e for each partitioning attribute, represented as:

$$\theta_o = \{(a_{p_1}, e_1), (a_{p_2}, e_2), (a_{p_3}, e_3), \ldots, (a_{p_n}, e_n)\}$$

With parallelism m, a partition function P is used to distribute elements among the state partitions. The k^{th} operator state partition can be represented as:

$$\theta_o^k = \{(a_{p_n}, e_n) \mid P(a_{p_n}) = k, (a_{p_n}, e_n) \in \theta_o\}, \\ \forall k \in \{1, 2, \ldots, m\}$$

Where:

- $P(a_{p_n}) = k$ indicates that the partition function maps a_{p_n} to state partition k.

IV. METHODOLOGY

In this section, we present the design of ScalaSSC. As illustrated in Fig. 1, the request processing workflow consists of four distinct phases. ScalaSSC consists of a centralized planner and multiple workers.

The planner is responsible for receiving requests from users or functions and dispatching them to workers. It incorporates a scheduler that manages information about hosts, operators, and state locations. Upon receiving a request, the planner first uses the scheduler to verify whether the invoked operator is stateless. If not, the scheduler identifies which state partition the request acts upon (state identification). Subsequently, the scheduler distributes this request to a specific worker (request distribution).

The worker carries out the processing of received requests inside executors. After receiving the request from the planner, the worker initially stores it in the queue. Periodically, a worker manager aggregates multiple queued requests into a batch request (batching). After that, the worker executes the batch request within a single executer/function (batch execution).

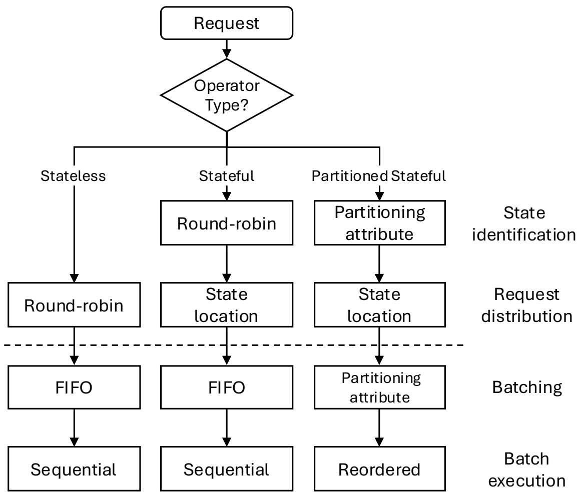


Figure 1: The workflow for ScalaSSC.

A. Request Scheduling Algorithm

Within the workflow of ScalaSSC, the first two phases are **state identification** and **request distribution**. Before running applications, the type of each operator and the parallelism of each stateful/partitioned stateful operator are predefined by users. Once configured, the scheduler creates each operator state partition sequentially and distributes it among workers until the desired parallelism is reached. The scheduler assigns each partition to the worker that has the fewest partitions belonging to the same operator as that partition. Ties are broken by selecting the worker with the fewest total partitions. Prior to batching, each request contains only a single tuple. For simplicity, we refer to the attributes of the request's sole tuple as the attributes of the request in §IV-A and §IV-B.

Our method caps the maximum number of in-flight requests at R_{max}. Upon receiving a request, the planner rejects it if the in-flight requests have already reached this limit; otherwise, the scheduler verifies the type of the invoked operator and schedules the request accordingly.

Request to stateless operator: The request is distributed across workers using a round-robin method.

Request to stateful operator: When the invoked operator encompasses multiple state partitions, the request acts upon only one state partition θ_o^i for processing. The scheduler assigns it to the k^{th} partition in a round-robin manner and routes it to the worker containing state partition θ_o^k.

Request to partitioned stateful operator: An request with partitioning attribute a_{p_i} acts upon element e_{p_i} for processing. The scheduler computes the k^{th} operator state partition contains element e_{p_i} by applying the partition function P (Virtual Nodes) to the attribute a_{p_i} and operator parallelism. Subsequently, it routes the request to the worker holding the state partition θ_o^k.

Then, the scheduler attaches the state partition θ_o^k to the request and dispatches it to the scheduled worker.

B. Request Queuing and Batching

The third phase in the workflow of ScalaSSC is **batching**. In our approach, each worker maintains multiple queues to store requests dispatched from the scheduler. The requests with the same invoked operator o and state partition θ_o^k are grouped in the same queue. It is worth noting that requests to stateless operators are grouped only by operator o.

A worker manager retrieves and batches requests from each queue in turn, and then forwards them to executors. To prevent resource overload, a maximum concurrency C_{max} is defined, representing the maximum number of executors allowed to execute concurrently for each queue. If there is no available executor, the manager will not retrieve any requests from it. The batch size B_{size} and batching window W are user-defined parameters for batching requests. The manager retrieves requests from the queue once the batch size B_{size} is reached or the window W expires, whichever occurs first. The strategies for retrieving vary in accordance with operator types:

Request to stateless operator and **stateful operator**: The worker manager enqueues requests into the queue according to their arrival time and retrieves requests in the FIFO order.

Request to partitioned stateful operator: Since a request acts upon only the state element associated with its partitioning attribute, the worker manager prioritizes retrieving requests with the same partitioning attribute from the queue to execute them together. This may disrupt the processing order of requests with different partitioning attributes; however, this does not affect the outcome, as they act upon different state elements.

Algorithm 1: PartitionedStateRequestsQueue

```
Data:
$b_e$: The expected processing batch index of enqueued
request, initialized as 1;
$M_{B_e,A_p}[R]$: A map where the key is the expected
batch index $b_e$ and the value is a map from the
partitioning attribute $a_p$ to a queue of requests.
1  Function Enqueue($r$):
2      $M_{b_e,r.a_p}$.push($r$)
3      if $M_{b_e}$.size == $B_{size}$ then
4          $b_e = b_e + 1$
5      end
6
7  Function Dequeue():
8      $D[R]$ = empty list of requests
9      $a_{prev}$ = $null$
10     while $D$.size less than $B_{size}$ do
11         $b_{min}$ = the minimum $b_e$ in $M$
12         if $a_{prev}$ != $null$ and $b_e$ of the earliest $r$ with
            $a_{prev}$ in $M - b_{min} < C_{max}$ then
13             $b_{e_{add}} = b_e$ of the earliest $r$ with $a_{prev}$
14             $r_{add} = M_{b_{e_{add}},a_{prev}}$.front
15         else
16             $r_{add}$ = the earliest request in $M_{b_{min}}$
17         end
18         $D$.add($r_{add}$)
19         $a_{prev}$ = $r_{add}.a_p$
20         $M$.remove($r_{add}$)
21         if $M$.size == 0 and $D$.size $< B_{size}$ then
22             $b_e = b_e + 1$
23             break
24         end
25     end
26     return $D$
```

As illustrated in Alg. 1, the key steps of the enqueue and dequeue operations on requests to partitioned stateful operator are outlined below. When receiving a request R with partitioning attribute $r.a_p$, the worker manager **Enqueue** it. We assign an expected processing batch index b_e to it and then push it to the queue belonging to $r.a_p$ within M_{b_e}(Line 2). The b_e assigned to the next request is incremented if the number of requests enqueued with the current expected batch index $M_{b_e}.size$ reaches the batch size (Lines 3–5). When retrieving requests, the manager uses **Dequeue** to select requests within a loop until the batch size is reached (Lines 10–25). For each selection (except the first request), we select the earliest request sharing the same partitioning attribute as the previously selected request if the difference between its expected batch index and the minimum expected batch index b_{min} in map M is less than C_{max} (Lines 11–14). Otherwise, the earliest request in M is selected (Line 16). This restriction ensures that no request experiences endless delay. Then, we add the selected request to the dequeued requests list D and update the previous attribute a_{prev} and the map M accordingly (Lines 18–20). If the map M becomes empty before the dequeued requests reach the batch size, indicating that these requests are dequeued upon expiration of the batch window W, we increment the expected batch number and terminate the loop (Lines 21–23). Finally, the dequeued requests are returned.

After retrieving requests, the worker manager combines their tuples into one batch request while preserving the original order and executes this batch request in an executor.

C. Batched Execution

API Function
θ_o **GetStateLock**()
Lock and get the operator state of current operator
void **SetStateUnlock**(θ_o)
Set and unlock the operator state of current operator
list[A_p, E] **GetParStateLock**(list[A_p])
Lock and get available elements of input partitioning attributes
void **SetParStateUnlock**(list[A_p, E])
Set and unlock elements of input list

Table I: Core API host interface.

In the **batch execution** phase of ScalaSSC, batch APIs are used to access states for multiple requests in one operation, aiming to reduce communication overhead. As depicted in Tab. I, the core batch APIs we support are: i) retrieving and setting the operator state; ii) getting the available, practically unlocked, state elements associated with the partitioning attributes from the input list and setting the updated elements back. By leveraging these APIs, we employ diverse execution strategies depending on the invoked operator:

Request to stateless operator and **stateful operator**: Request's tuples are processed sequentially, while storing the chained calls temporarily. Once all the tuples have been processed, the chained calls are sent to the planner. However, stateful operators must use `GetStateLock` to read the operator state before processing tuples and `SetStateUnlock` to update the operator state after processing all the tuples (before sending chain calls).

Request to partitioned stateful operator: Request's tuple acts upon only the state element related to its partitioning attribute. Thus, accessing the entire operator state results in unnecessary communication and state contention overhead. Within the execution of a request, the executor processes tuples within a loop. In every loop, the executor uses `GetParStateLock` to acquire available elements acted upon by unprocessed tuples. Afterward, the executor processes tuples whose acting-upon elements have been acquired sequentially. Later on, it uses `SetParStateUnlock` to write updated elements back. Within each loop, all chained calls are temporarily stored. The loop iterates through multiple rounds until all tuples are processed. Finally, the stored chained calls are sent.

The function `GetParStateLock` operates by initially making an attempt to acquire locks for all the unlocked partitioning attributes within the input list. Once any lock has been obtained, it will return the elements related to the acquired attributes. Otherwise, it registers and waits for all attributes within the input list. Whenever any attribute is released, it attempts to lock those attributes once again.

V. Performance Evaluation

In this section, we evaluate the following aspects of ScalaSSC: (i) the benefits of co-locating requests with states they act upon (§ IV-A), (ii) the performance improvements from increasing operator parallelism(§ IV-A), (iii) the gains achieved by batching requests(§ IV-C) and executing requests that act upon the same state together(§ IV-B), and (iv) the overall performance improvements compared to serverless baselines.

A. Experiment setup

a) Serverless Baselines and Testbed: We compared ScalaSSC against FAASM and AWS Lambda. In FAASM, a request queue has been added to the planner. Otherwise, chained call requests might be lost. Both FAASM and ScalaSSC are executed on the same Kubernetes cluster consisting of 4 nodes. The Planner is deployed on an 8-core AMD EPYC 2.30 GHz machine with 1500GB of RAM to load input data and collect performance metrics in memory. Three worker nodes are run on 4-core AMD EPYC 2.30 GHz machines with 16GB of RAM. In the Lambda setup, each function is configured without a concurrency limitation and allocated 1769 MB of memory, corresponding to one full vCPU [24]. We use Kinesis [23] for transferring intermediate messages and batching requests, and ElastiCache [22], which offers the lowest access latency among AWS storage options [16], to store state. Each Kinesis stream is initialized in on-demand mode, which automatically scales throughput up to 200,000 records per second. Our ElastiCache setup utilizes the Serverless Redis OSS Cache.

b) Evaluation Metrics: Our primary goal is to increase throughput and reduce end-to-end latency. We define *throughput* as the number of tuples processed end-to-end (i.e., by all operators) per second, while *end-to-end latency* refers to the time it takes for a tuple and all of its subtuples to be processed by the entire pipeline. We also evaluate the *average tuple processing latency* for non-stateless operators. This is defined as the time it takes to retrieve the initial state, process tuples, and write all updated states back, divided by the number of processed tuples.

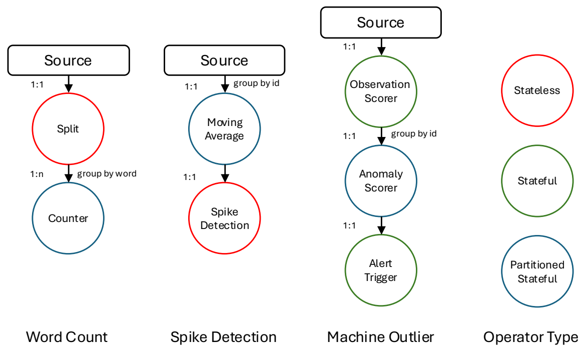


Figure 2: The Workflows of WC, SD, and MO. The ratios of the input tuples between different operators are marked.

c) Benchmark Applications: We evaluate our approach on Word Count (WC), Spike Detection (SD), and Machine Outlier (MO) applications from DSPBench [6], whose workflows are shown in Fig 2. Their input data are sourced from real-world datasets. In WC, we measure throughput based on the number of words counted rather than the number of sentences. In MO, as the first operator aggregates tuples and subsequently processes them collectively when a threshold is reached, not every tuple has subtuples processed by the subsequent operators. We therefore measure the end-to-end latency based only on the tuples whose subtuples are processed by all operators. It is important to note that executing requests in concurrent executors does not guarantee the processing order will align with the receiving order, and this discrepancy should be tolerated within the application logic.

This work does not address the optimization of any parameter for performance enhancement. In ScalaSSC, we configure the maximum in-flight requests R_{max} as 15,000 and batch window W as 20 ms. By default, we set the batch size to 20, the concurrency to 10, the parallelism for stateful operators and partitioned stateful operators to 1 and 3, respectively. Our evaluation reports end-to-end latency in milliseconds and compares the 99th percentile latency. The average tuple processing latency is measured in nanoseconds, and the average values are reported. Each experiment is run for ten minutes and five times.

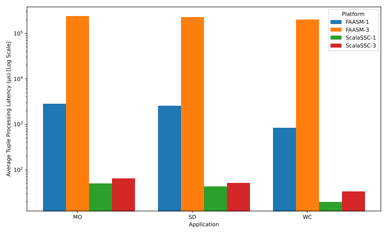


Figure 3: Comparison of the tuple processing latency for FAASM and ScalaSSC with varied nodes/parallelism.

B. State Access Evaluation

This experiment demonstrates state access latency improvements achieved by collocating requests with the states they act upon. We executed the WC, SD, and MO applications on FAASM by using a round-robin scheduler with both single-worker and three-worker configurations. Besides, with batch size 1, we conducted these applications on ScalaSSC using parallelism hints of one and three for the partitioned stateful operators. With FAASM configured to use a single worker and ScalaSSC set to parallelism of one, all requests to partitioned stateful operator run on the same worker. However, with FAASM using three workers and ScalaSSC with three parallelism hints, all requests are executed across three workers. In each operator, the processing logic for each tuple remains the same, and the average tuple processing latency differences across setups are predominantly determined by state access time. For both FAASM and ScalaSSC, the input tuple rate is set at 1 tuple per second. Additionally, the systems process only one input tuple concurrently to ensure that the impact of lock contention on the state access time stays consistent across all configurations. We compare the average tuple processing latency of partitioning stateful operators.

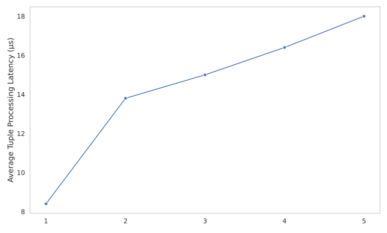
As shown in Fig.3, in WC, FAASM's average tuple processing latency with one worker (850 μs) is significantly lower than with three workers (202,384 μs). In ScalaSSC, the latency with parallelism hint 1 (20 μs) is nearly identical to that with parallelism hint 3 (34 μs). SD and MO exhibit similar behavior.

In FAASM, increasing the cluster workers from one to three introduces remote state access, resulting in a significant increase in latency. However, in ScalaSSC, the latency remains unchanged when additional workers are utilized. Because the tuple and the state it acts upon are co-located in the same worker, there is no introduction of cross-worker communication. The latency of ScalaSSC in parallelism 1 is lower than that of FAASM in one worker. This is because FAASM utilizes Redis to maintain the location of the master copy for every state, introducing high latency when initializing the state. In contrast, our approach uses consistent hashing to identify the state location, eliminating the high Redis access latency.

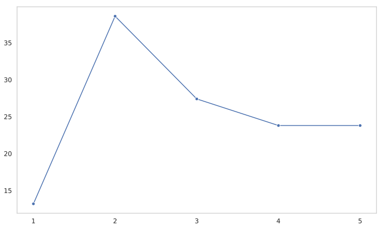
C. Request Grouping Evaluation

This experiment investigates the impact of executing tuples act upon the same state element together in ScalaSSC. In the WC, SD, and MO applications, we set the batch size to 30 and the parallelism hint for every operator to one, varying the concurrency from 1 to 5 to compare the average tuple processing latency of partitioned operators.

As shown in Fig. 4, the average tuple processing latency increases by 64%, 192%, and 62% for WC, SD, and MO, respectively, when concurrency rises from 1 to 2. After which, in WC, the latency grows slowly with further increases in concurrency. In contrast, in SD and MO, latency decreases by 38% and 5%, respectively, at concurrency 5 compared to concurrency 2.



(a) WC


(b) SD

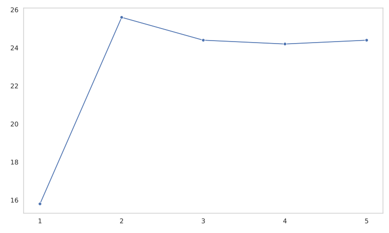
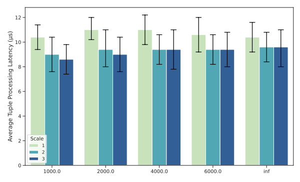
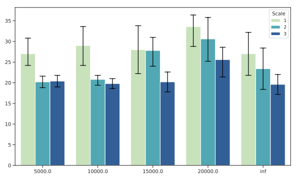

(c) MO

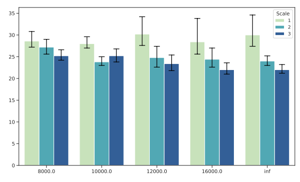
Figure 4: Comparison of tuple processing latency for ScalaSSC under varied concurrency. The X-axis represents concurrency.



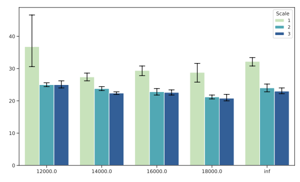
(a) WC p-latency



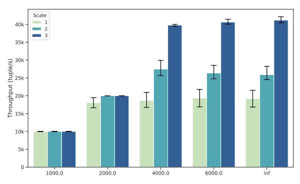
(b) SD p-latency



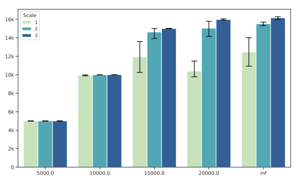
(c) MO p-latency



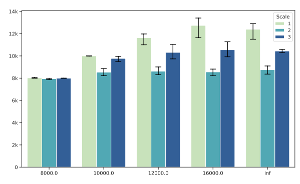
(d) MO-5 p-latency



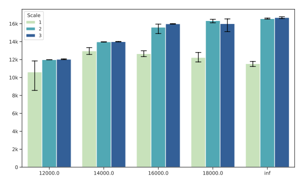
(e) WC throughput



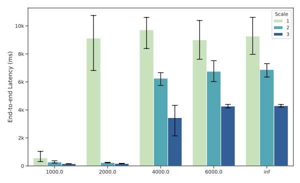
(f) SD throughput



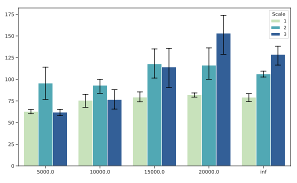
(g) MO throughput



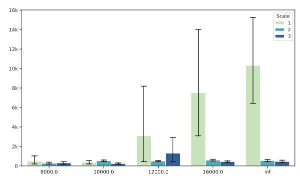
(h) MO-5 throughput



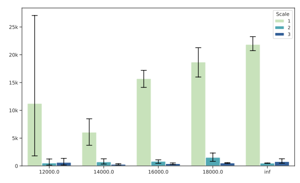
(i) WC E2E latency



(j) SD E2E latency



(k) MO E2E latency



(l) MO-5 E2E latency

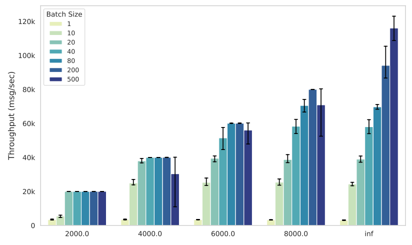
Figure 5: Comparison of tuple processing latency (p-latency), throughput and end-to-end latency metrics (E2E latency) in ScalaSSC with varied parallelism under different input rates (tuple/s). The X-axis represents input rates.

The average tuple processing latency increases when the concurrency level changes from 1 to 2 due to the introduction of state access contention, resulting in lock waiting time and increased state access operations. As concurrency increases further, latency in SD and MO decreases because higher concurrency offers greater flexibility in prioritizing tuples with the same partitioning attribute. Executing tuples that act upon the same state element together not only reduces lock contention between executors, but also enables loaded elements to be acted upon by multiple tuples. However, WC does not exhibit this trend. This discrepancy arises from differences in the partitioning attributes of their partitioned operators: the *counter* of WC uses words, whereas the *moving average* of SD and *anomaly scorer* of MO use sensor or machine IDs. The number of unique words in the dataset is significantly higher than the number of IDs (approximately 100 unique IDs), resulting in a lower likelihood of combining tuples with the same partitioning attribute together.

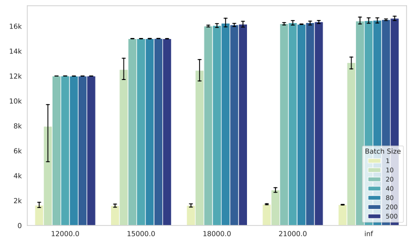
D. Parallelism

This experiment investigates the impact of increasing operator parallelism. We vary the parallelism of partitioned operators and the input rate for WC, SD and MO applications to compare end-to-end latency, throughput, and average tuple processing latency of partitioned operators under different parallelism levels. In the following evaluations, input rates were incrementally increased until unthrottled, denoted as "inf" in the figures. Additionally, MO was executed on a cluster with 5 workers.

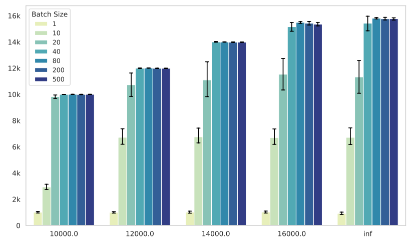
As shown in Figs.5a–5d, in WC, increasing the parallelism hints to 2 and 3 reduces the average tuple processing latency of the partitioned operator by up to 18% and 15%, respectively. SD and MO exhibit comparable behavior. The reduction in latency occurs because fewer state elements are stored in each



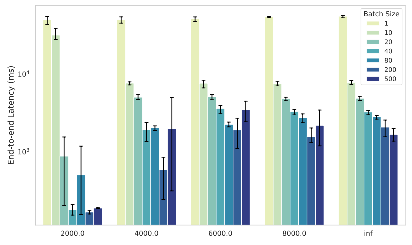
(a) WC throughput



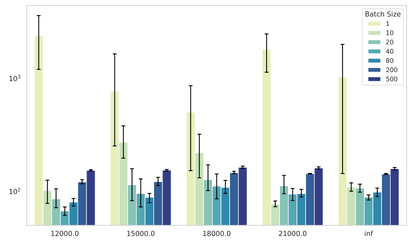
(b) SD throughput



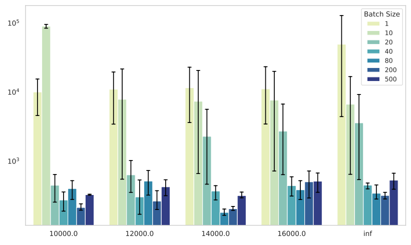
(c) MO throughput



(d) WC latency



(e) SD latency



(f) MO latency

Figure 6: Comparison of throughput and end-to-end latency (E2E latency) in ScalaSSC with varied batch size under different input rates (tuple/s). The X-axis represents input rates.

state server after scaling, which increases the likelihood of executing tuples with the partitioning attribute in batch.

As shown in Figs.5e–5h, parallelism hints 2 and 3 achieve up to 47% and 115% higher throughput in WC. Similar results are achieved in SD and MO with 5 workers. However, there is a degradation in throughput of up to 32% and 17% in MO at parallelism hints 2 and 3 respectively. In WC and SD, higher parallelism achieves greater throughput by reducing average tuple processing latency and distributing the workload among other workers so as to utilize available resources on other nodes.

MO is made up of one partitioned stateful operator and two stateful operators. When the parallelism is set to 1 and there are three workers, it distributes the states of the three operators as well as their corresponding requests evenly among the workers, with one state being placed in each worker. After scaling the parallelism of the partitioned stateful operator to 2, its scaled state partition and corresponding requests are distributed to another node, co-located with another operator state. Due to the resource contention, this operator quickly becomes the bottleneck and decreases the throughput. After scaling parallelism to 3, the workload on this worker is reduced, resulting in higher throughput. In the cluster with 5 workers, all the scaled state partitions of MO will be distributed to a new worker, increasing the throughput by using more available resources.

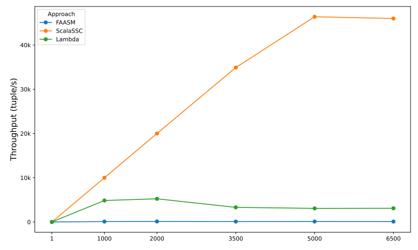
As depicted in Figs. 5i–5l, with the maximum input rate, when parallelism hints are set to 2 and 3, the end-to-end latency results in a 25% and 52% reduction for WC respectively, a 34% and 62% increase for SD respectively, and a 95% and 96% reduction for MO respectively. In WC and MO, higher parallelism reduces latency not only by decreasing the tuple processing time but also by reducing request waiting time for available executors through increased capacity of the scaled operator. MO achieves significant latency reduction because the *observation scorer* operator aggregates requests, which results in a peak input flow to the successor operator. The scaled operator with higher capacity can handle this peak more effectively.

In SD, the first operator is scaled. As the end-to-end latency is calculated starting from the moment when the tuple is being processed by the first operator, the reduced executor waiting time of the first operator is not reflected in latency. Additionally, both the increased communication overhead that occurs when communicating with more workers and the higher throughput resulting from the higher parallelism contribute to an increase in latency.

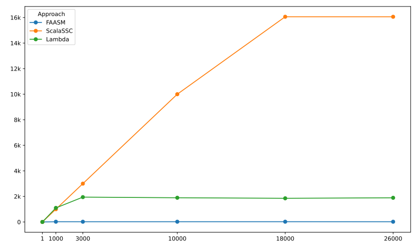
E. Batch Execution Evaluation

This experiment demonstrates the performance improvements achieved by batching. We varied the batch size and input rate for the WC, SD and MO applications to compare performance under different batch sizes.

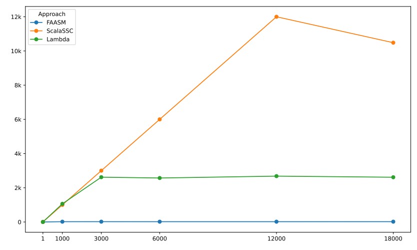
As shown in Figs. 6a–6c, within maximum input range, by batching requests, WC, SD, and MO achieve throughput increase up to 36 times, 9 times, and 16 times when compared to the throughput at batch size of 1. WC achieves maximum throughput at batch size 500, whereas SD and MO peak at batch sizes of 20 and 40, respectively. The throughput is increased via batching as requests executed together share the initialized executor as well as the loaded state, reducing the relevant workload. Compared to WC, both SD and MO achieve maximum throughput at smaller batch sizes. The reason for



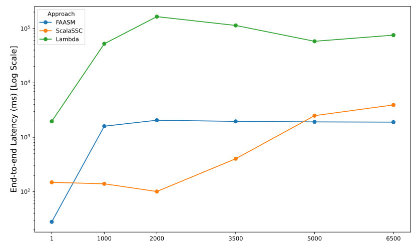
(a) WC throughput



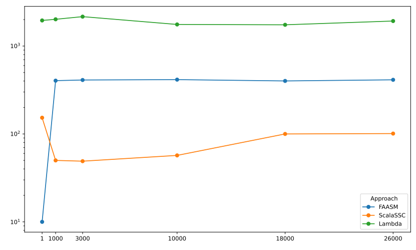
(b) SD throughput



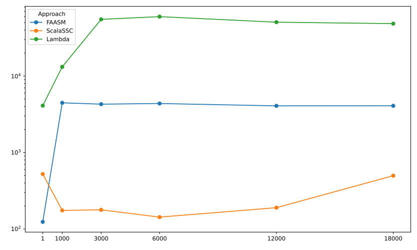
(c) MO latency



(d) WC latency



(e) SD latency



(f) MO latency

Figure 7: Comparison of throughput and end-to-end latency metrics for FAASM, Lambda and ScalaSSC under different input rates (tuple/s). The X-axis represents input rates.

this is that WC is a data-intensive application, while SD and MO are compute-intensive applications, and their throughput is restricted by the computation workload.

As shown in Figs. 6d–6f, increasing batch sizes initially decreases end-to-end latency, but beyond a certain batch size, the latency begins to rise. For different input rates, the minimum latency is achieved at different batch sizes. Initially, latency decreases as larger batch sizes provide higher throughput capacity for processing tuples, reducing the waiting time for available executors. However, as batch sizes grow, the inputs fail to fill the batch within the window time, leading to unnecessary waiting time.

F. Overall Evaluation

This experiment compares the performance between FAASM, Lambda and ScalaSSC. We run WC, SD, and MO under varying input rates to benchmark ScalaSSC. As illustrated in Fig. 7, within the maximum input rate range, our approach exhibits a throughput enhancement of up to 845 times compared to FAASM and 14 times compared to Lambda respectively. Additionally, it delivers a reduction in end-to-end latency of up to 87% and 95% respectively.

Compared with FAASM, ScalaSSC shows higher latency when the input rate is 1. This is mainly because of the time to accumulate a batch request. ScalaSSC shows higher throughput than FAASM because of the use of batch processing. Moreover, it achieves lower latency under high input rates by reducing the state access latency and available executors waiting time. Furthermore, our approach maintains end-to-end latency at the millisecond level, whereas Lambda operates at the second level, due to the integration of state servers within workers and the support for batch state access APIs.

VI. Conclusions and Future Work

In this paper, we propose ScalaSSC, a stateful serverless computing framework designed for stream processing applications. It co-locates the request with the state it acts upon and increases operator parallelism by scaling the state instance, aiming to reduce state access latency. In batch execution, it executes the requests act upon the same state together and provides batch APIs to enhance the efficiency of state access. Experimental results show the enhancements in both throughput and latency. As shown in the experiment, parameters like batch size and parallelism have a significant impact on performance. Our future direction is dynamically optimizing these parameters during the runtime to achieve better performance.

References

[1] A. Ali, R. Pinciroli, F. Yan, and E. Smirni. Batch: Machine learning inference serving on serverless platforms with adaptive batching. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2020.

[2] A. Ali, R. Pinciroli, F. Yan, and E. Smirni. Optimizing inference serving on serverless platforms. *Proc. VLDB Endow.*, 15(10):2071–2084, June 2022.

[3] L. Ao, L. Izhikevich, G. M. Voelker, and G. Porter. Sprocket: A serverless video processing framework. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, page 263–274, New York, NY, USA, 2018. Association for Computing Machinery.

[4] D. Barcelona-Pons, P. García-López, and B. Metzler. Glider: Serverless ephemeral stateful near-data computation. In *Proceedings of the 24th International Middleware Conference*, pages 247–260, 2023.

[5] D. Barcelona-Pons, P. Sutra, M. Sánchez-Artigas, G. París, and P. García-López. Stateful serverless computing with crucial. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 31(3):1–38, 2022.

[6] M. V. Bordin, D. Griebler, G. Mencagli, C. F. Geyer, and L. G. L. Fernandes. Dspbench: A suite of benchmark applications for distributed data stream processing systems. *IEEE Access*, 8:222900–222917, 2020.

[7] N. L. Bragazzi, H. Dai, G. Damiani, M. Behzadifar, M. Martini, and J. Wu. How big data and artificial intelligence can help better manage the covid-19 pandemic. *International Journal of Environmental Research and Public Health*, 17(9):3176, 2020.

[8] S. Cai, Z. Zhou, K. Zhao, and X. Chen. Cost-efficient serverless inference serving with joint batching and multi-processing. In *Proceedings of the 14th ACM SIGOPS Asia-Pacific Workshop on Systems*, APSys '23, page 43–49, New York, NY, USA, 2023. Association for Computing Machinery.

[9] Z. Cai, Z. Chen, X. Chen, R. Ma, H. Guan, and R. Buyya. Spsc: Stream processing framework atop serverless computing for industrial big data. *IEEE Transactions on Cybernetics*, 2024.

[10] C. Cicconetti, M. Conti, and A. Passarella. A decentralized framework for serverless edge computing in the internet of things. *IEEE Transactions on Network and Service Management*, 18(2):2166–2180, 2020.

[11] R. Eidenbenz and T. Locher. Task allocation for distributed stream processing. In *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*, pages 1–9, 2016.

[12] B. Gedik, S. Schneider, M. Hirzel, and K.-L. Wu. Elastic scaling for data stream processing. *IEEE Transactions on Parallel and Distributed Systems*, 25(6):1447–1463, 2013.

[13] A. Hall and U. Ramachandran. An execution model for serverless functions at the edge. In *Proceedings of the International Conference on Internet of Things Design and Implementation*, pages 225–236, 2019.

[14] S. Han, N. Egi, A. Panda, S. Ratnasamy, G. Shi, and S. Shenker. Network support for resource disaggregation in next-generation datacenters. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*, pages 1–7, 2013.

[15] M. H. Iqbal and T. R. Soomro. Big data analysis: Apache storm perspective. *International journal of computer trends and technology*, 19(1):9–14, 2015.

[16] A. Khandelwal, Y. Tang, R. Agarwal, A. Akella, and I. Stoica. Jiffy: Elastic far-memory for stateful serverless analytics. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 697–713, 2022.

[17] W. Li, Y. Liang, and S. Wang. *Data driven smart manufacturing technologies and applications*. Springer, 2021.

[18] Z. Li, Y. Liu, L. Guo, Q. Chen, J. Cheng, W. Zheng, and M. Guo. Faasflow: Enable efficient workflow execution for function-as-a-service. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 782–796, 2022.

[19] D. Mvondo, M. Bacou, K. Nguetchouang, L. Ngale, S. Pouget, J. Kouam, R. Lachaize, J. Hwang, T. Wood, D. Hagimont, et al. Ofc: an opportunistic caching system for faas platforms. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pages 228–244, 2021.

[20] S. Nastic, T. Rausch, O. Scekic, S. Dustdar, M. Gusev, B. Koteska, M. Kostoska, B. Jakimovski, S. Ristov, and R. Prodan. A serverless real-time data analytics platform for edge computing. *IEEE Internet Computing*, 21(4):64–71, 2017.

[21] A. P. Rodrigues, R. Fernandes, A. Bhandary, A. C. Shenoy, A. Shetty, and M. Anisha. Real-time twitter trend analysis using big data analytics and machine learning techniques. *Wireless Communications and Mobile Computing*, 2021(1):3920325, 2021.

[22] A. W. Services. Amazon elasticache. https://aws.amazon.com/elasticache/, 2024.

[23] A. W. Services. Amazon kinesis. https://aws.amazon.com/kinesis/, 2024.

[24] A. W. Services. Configure lambda function memory. https://docs.aws.amazon.com/lambda/latest/dg/configuration-memory.html, 2024.

[25] S. Shillaker and P. Pietzuch. Faasm: Lightweight isolation for efficient stateful serverless computing. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 419–433, 2020.

[26] W. W. Song, T. Um, S. Elnikety, M. Jeon, and B.-G. Chun. Sponge: Fast reactive scaling for stream processing with serverless frameworks. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 301–314, Boston, MA, July 2023. USENIX Association.

[27] V. Sreekanti, C. Wu, X. C. Lin, J. Schleier-Smith, J. E. Gonzalez, J. M. Hellerstein, and A. Tumanov. Cloudburst: stateful functions-as-a-service. *Proc. VLDB Endow.*, 13(12):2438–2452, July 2020.

[28] E. Volnes, T. Plagemann, and V. Goebel. To migrate or not to migrate: An analysis of operator migration in distributed stream processing. *IEEE Communications Surveys & Tutorials*, 2023.

[29] H. Wang, D. Niu, and B. Li. Distributed machine learning with a serverless architecture. In *Proceedings of the IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*, pages 1288–1296, 2019.

[30] Z. Wu, Y. Deng, Y. Zhou, L. Cui, and X. Qin. Hashcache: Accelerating serverless computing by skipping duplicated function execution. *IEEE Transactions on Parallel and Distributed Systems*, 2023.

[31] Z. Wu, Y. Deng, Y. Zhou, J. Li, S. Pang, and X. Qin. Faasbatch: Boosting serverless efficiency with in-container parallelism and resource multiplexing. *IEEE Transactions on Computers*, 73(4):1071–1085, 2024.

[32] S. Yang. Iot stream processing and analytics in the fog. *IEEE Communications Magazine*, 55(8):21–27, 2017.

[33] Y. Yang, L. Zhao, Y. Li, H. Zhang, J. Li, M. Zhao, X. Chen, and K. Li. Infless: a native serverless system for low-latency, high-throughput inference. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 768–781, New York, NY, USA, 2022. Association for Computing Machinery.

[34] X. Yao, N. Chen, X. Yuan, and P. Ou. Performance optimization of serverless edge computing function offloading based on deep reinforcement learning. *Future Generation Computer Systems*, 139:74–86, 2023.

[35] C. Zhang, M. Yu, W. Wang, and F. Yan. MArk: Exploiting cloud services for Cost-Effective, SLO-Aware machine learning inference serving. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 1049–1062, Renton, WA, July 2019. USENIX Association.