GrandSLAm: Guaranteeing SLAs for Jobs in Microservices Execution Frameworks

Ram Srivatsa Kannan
University of Michigan, Ann Arbor
ramsri@umich.edu

Lavanya Subramanian*
Facebook
lavanya.subramanian@oculus.com

Ashwin Raju
University of Texas at Arlington
ashwin.raju93@mavs.uta.edu

Jeongseob Ahn†
Ajou University
jsahn@ajou.ac.kr

Jason Mars
University of Michigan, Ann Arbor
profmars@umich.edu

Lingjia Tang
University of Michigan, Ann Arbor
lingjia@umich.edu

Abstract

The microservice architecture has dramatically reduced user effort in adopting and maintaining servers by providing a catalog of functions as services that can be used as building blocks to construct applications. This has enabled datacenter operators to look at managing datacenter hosting microservices quite differently from traditional infrastructures. Such a paradigm shift calls for a need to rethink resource management strategies employed in such execution environments. We observe that the visibility enabled by a microservices execution framework can be exploited to achieve high throughput and resource utilization while still meeting Service Level Agreements, especially in multi-tenant execution scenarios.

In this study, we present GrandSLAm, a microservice execution framework that improves utilization of datacenters hosting microservices. GrandSLAm estimates time of completion of requests propagating through individual microservice stages within an application. It then leverages this estimate to drive a runtime system that dynamically batches and reorders requests at each microservice in a manner where individual jobs meet their respective target latency while achieving high throughput. GrandSLAm significantly increases throughput by up to 3× compared to the our baseline, without violating SLAs for a wide range of real-world AI and ML applications.

CCS Concepts • **Software and its engineering → Software as a service orchestration system**;

Keywords Microservice, Systems and Machine Learning

ACM Reference Format:
Ram Srivatsa Kannan, Lavanya Subramanian, Ashwin Raju, Jeongseob Ahn, Jason Mars, and Lingjia Tang. 2019. GrandSLAm: Guaranteeing SLAs for Jobs in Microservices Execution Frameworks. In *Proceedings of Fourteenth EuroSys Conference 2019 (EuroSys '19).* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3302424.3303958

*This work was done while the author worked at Intel Labs
†Corresponding author

EuroSys '19, March 25–28, 2019, Dresden, Germany

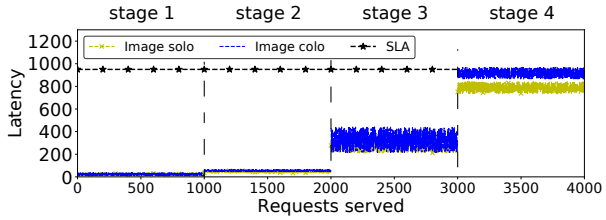
ACM ISBN 978-1-4503-6281-8/19/03…$15.00
https://doi.org/10.1145/3302424.3303958

1 Introduction

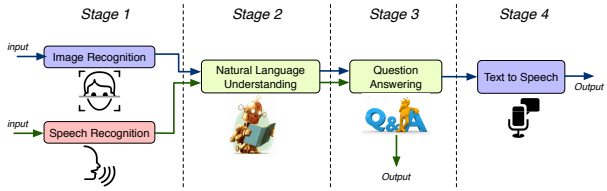
The microservice architecture along with cloud computing is dramatically changing the landscape of software development. A key distinguishing aspect of the microservice architecture is the availability of pre-existing, well-defined and implemented software services by cloud providers. These microservices can be leveraged by the developer to construct their applications without perturbing the underlying hardware or software requirements. The user applications can, therefore, be viewed as an amalgamation of microservices. The microservice design paradigm is widely being utilized by many cloud service providers driving technologies like Serverless Computing [3, 5, 13, 14, 19, 20].

Viewing an application as a series of microservices is helpful especially in the context of datacenters where the applications are known ahead of time. This is in stark contrast to the traditional approach where the application is viewed as one monolithic unit and instead, lends a naturally segmentable structure and composition to applications. Such behavior is clearly visible for applications constructed using artificial intelligence and machine learning (AI and ML) services, an important class of datacenter applications which has been leveraging the microservice execution framework. As a result, it opens up new research questions especially in the space of multi-tenant execution where multiple jobs, applications or tenants share common microservices.

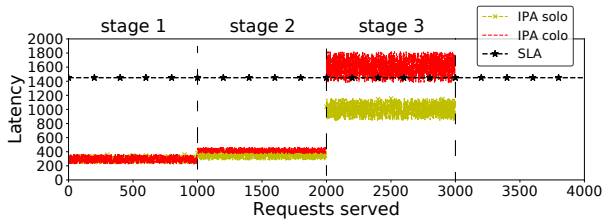
Multi-tenant execution has been explored actively in the context of traditional datacenters and cloud computing frameworks towards improving resource utilization [10, 31, 41, 48]. Prior studies have proposed to co-locate high priority latency-sensitive applications with other low priority batch applications [31, 48]. However, the multi-tenant execution



(a) Image Querying: SLA is not violated



(b) Sharing NLU and QA microservices



(c) IPA: SLA is violated

Figure 1. Sharing the two common microservices between Image Querying and Intelligent Personal Assistant applications

in a microservice based computing framework would operate on a fundamentally different set of considerations and assumptions since resource sharing can now be viewed at a microservice granularity rather than at an entire application granularity.

Figure 1b illustrates an example scenario in which an end-to-end Intelligent Personal Assistant (IPA) application shares the `Natural Language Understanding` (NLU) and `Question Answering` (QA) microservices with an image based querying application. Each of these applications is constructed as an amalgamation of different microservices (or stages). In such a scenario, the execution load in these particular microservices increases, thereby causing the latency of query execution in stages 2 and 3 to increase. This increase in latency at specific stages affects the end-to-end latency of the IPA application, thereby violating service level agreements (SLAs). This phenomenon is illustrated by Figure 1c and Figure 1a. The x-axis represents the number of requests served while the y-axis denotes latency. Horizontal dotted lines separate individual stages. As can be seen, the SLA violation for the image querying application in Figure 1a is small, whereas the IPA application suffers heavily from SLA violation. However, our understanding of the resource contention need not stop at such an application granularity, unlike traditional private data centers. It can rather be broken down into contention at the microservice granularity, which makes resource contention management a more tractable problem.

This fundamentally different characteristic of microservice environments motivates us to rethink the design of runtime systems that drive multi-tenancy in microservice execution frameworks. Specifically, in virtualized datacenters, consolidation of multiple latency critical applications is limited, as such scenarios can be performance intrusive. In particular, the tail latency of these latency critical applications could increase significantly due to the inter-application interference from sharing the hardware resources [31, 32, 48, 51]. Even in a private datacenter, there is limited visibility into application specific behavior and SLAs, which makes it hard even to determine the existence of such performance intrusion [27]. As a result, cloud service providers would not be able to meet SLAs in such execution scenarios that co-locate multiple latency critical applications. In stark contrast, the execution flow of requests through individual microservices is much more transparent.

We observe that this visibility creates a new opportunity in a microservice-based execution framework and can enable high throughput from consolidating the execution of multiple latency critical jobs, while still employing fine-grained task management to prevent SLA violations. In this context, satisfying end-to-end SLAs merely becomes a function of meeting disaggregated partial SLAs at each microservice stage through which requests belonging to individual jobs propagate. However, focusing on each microservice stage's SLAs standalone misses a key opportunity, since we observe that there is significant variation in the request level execution slack among individual requests of multiple jobs. This stems from the variability that exists with respect to user specific SLAs, which we seek to exploit.

In this study, we propose GrandSLAm, a holistic runtime framework that enables consolidated execution of requests belonging to multiple jobs in a microservice-based computing framework. GrandSLAm does so by providing a prediction based on identifying `safe` consolidation to deliver satisfactory SLA (latency) while maximizing throughput simultaneously. GrandSLAm exploits the microservice execution framework and the visibility it provides especially for AI and ML applications, to build a model that can estimate the completion time of requests at different stages of a job with high accuracy. It then leverages the prediction model to estimate per-stage SLAs using which it (1) ensures end-to-end job latency by reordering requests to prioritize those requests with low computational slack, (2) batches multiple requests to the maximum extent possible to achieve high throughput under the user specified latency constraints. It is important to note that employing each of these techniques standalone does not yield SLA enforcement. An informed combination

of request re-ordering with a view of end-to-end latency slack and batching is what yields effective SLA enforcement, as we demonstrate later in the paper. Specifically, this paper makes the following contributions:

- **Analysis of microservice execution scenarios.** Our investigation observes the key differences between traditional and microservice-based computing platforms – primarily in the context of visibility into the underlying microservices that provide exposure to application specific SLA metrics.
- **Accurate estimation of completion time at individual microservice stages.** We build a model that estimates the completion time of individual requests at the different microservice stages and hence, the overall time of completion. We have demonstrated high accuracy in estimating completion times, especially for AI and ML microservices.
- **Guarantee end-to-end SLAs by exploiting stage level SLAs.** By utilizing the completion time predictions from the model, we derive individual stage SLAs for each microservice/stage. We then combine this per-stage SLA requirement with our understanding of end-to-end latency and slack. This enables an efficient request scheduling mechanism towards the end goal of maximizing server throughput without violating the end-to-end SLA.

Our evaluations on a real system deployment of a 6 node CPU cluster coupled with graphics processing accelerators demonstrates GrandSLAm's capability to increase the throughput of a datacenter by up to 3× over the state-of-the-art request execution schemes for a broad range of real-world applications. We perform scale-out studies as well that demonstrate increased throughput while meeting SLAs.

2 Background

In this section, we first describe the software architecture of a typical microservice and its execution framework. We then describe unique opportunities a microservice framework presents as compared to a traditional datacenter, for an efficient redesign.

2.1 Microservices Software Architecture

The microservice architecture is gaining popularity among software engineers, since it enables easier application development and deployment while not having to worry about the underlying hardware and software requirements. Microservices resemble well-defined libraries that perform specific functions, which can be exposed to consumers (i.e., application developers) through simple APIs. With the microservice paradigm approach, instead of writing an application from scratch, software engineers leverage these microservices as building blocks to construct end-to-end applications. The end-to-end applications consist of a chain of microservices many of which are furnished by the datacenter service providers. Microservice based software architectures speed up deployment cycles, foster application-level innovation by providing a rich set of primitives, and improve maintainability and scalability, for application classes where the same building blocks tend to be used in many application contexts [13].

Traditional, multi-tier architectures compartmentalize application stages based on the nature of services into different tiers. In most cases, application stages belong to either the presentation layer which focuses on the user interface, application processing layer in which the actual application execution occurs and the data management layer which stores data and metadata belonging to the application. This is fundamentally different from the microservice architecture. Microservices, at each stage in a multi-stage application, perform part of the processing in a large application. In other words, one can imagine a chain of microservices to constitute the application processing layer.

2.2 Microservices Use Cases

With the advent of **Serverless Computing** design, the microservices paradigm is being viewed as a convenient solution for building and deploying applications. Several cloud service providers like Amazon (AWS Lambda [3]) and IBM (IBM Bluemix [29]) utilize the microservice paradigm to offer services to their clients. Typically, microservices hosted by cloud service providers provide the necessary functionality for each execution stage in every user's multi-stage application. In this context, a motivating class of applications that would benefit from the microservice paradigm is artificial intelligence (AI) and machine learning (ML) applications [39]. Many of the stages present in the execution pipeline of AI applications are common across other AI applications [13]. As shown in the example in Figure 1b, a speech-input based query execution application is constructed as an amalgamation of microservices that performs speech recognition, natural language understanding, and a question answering system. Similarly, an image-input based query system/application also uses several of these same microservices as its building blocks.

FaaS (Function-as-a-Service) or Serverless based cloud services contain APIs to define the workflow of a multi-stage application as a series of steps representing a Directed Acyclic Graph (DAG). For instance, some of the workflow types (DAGs) that are provided by Amazon as defined by AWS step functions [4] are shown in Figure 2. Elgamal et al. talk about this in detail [11]. Figure 2 (a) shows the simplest case where the DAG is sequential. From our study, we were able to find that several real-world applications (Applications Table 3) and customers utilizing AWS Lambda possess workflow DAGs there were sequential. Figure 2 (b) shows a workflow DAG with parallel steps in which multiple functions are executed in parallel, and their outputs are aggregated before the next function starts. The last type of

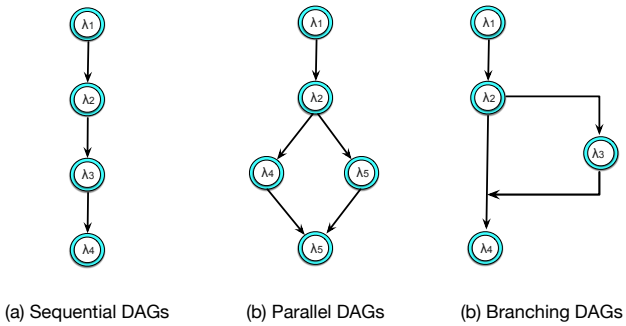
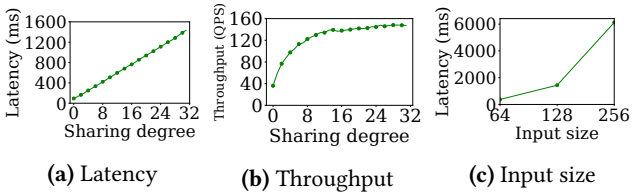


Figure 2. Types of DAGs used in applications based on microservices



(a) Latency **(b)** Throughput **(c)** Input size

Figure 3. Increase in latency, throughput, and input size as the sharing degree increases

workflow DAGs possesses branching steps shown in Figure 2 (c). Such workflows typically have a branch node that has a condition to decide in which direction the branch execution would proceed. In our paper, we focus only on sequential workflows as shown in Figure 2 (a). In Section 5, we will discuss the limitation of our study and possible extensions for the complex workflows.

2.3 Challenges

Although the usage and deployment of microservices are fundamentally different from traditional datacenter applications, the low resource utilization problem persists even in datacenters housing microservices [15, 16, 40]. In order to curb this, datacenter service providers could potentially allow sharing of common microservices across multiple jobs as shown in Figure 1b. However, these classes of applications, being user-facing, are required to meet strict Service Level Agreements (SLAs) guarantees. Hence, sharing microservices could create contention, resulting in the violation of end-to-end latency of individual user-facing applications, thereby violating SLAs. This is analogous to traditional datacenters where there is a tendency to actively avoid co-locating multiple user-facing applications, leading to over-provisioning of the underlying resources when optimizing for peak performance [6].

2.4 Opportunities

However, the microservice execution environments fundamentally change several operating assumptions present in traditional datacenters that enable much more efficient multi-tenancy, while still achieving SLAs. First, the microservice execution framework enables a new degree of visibility into an application's structure and behavior, since an application is comprised of microservice building blocks. This is different from traditional datacenter applications where the application is viewed as one monolithic unit. Hence, in such a traditional datacenter, it becomes very challenging to even identify, let alone prevent interference between co-running applications [27, 31, 48]. Second, the granularity of multi-tenancy and consolidation in a microservice framework is distinctively different from traditional datacenter systems. Application consolidation in microservice execution platforms is performed at a fine granularity, by batching multiple requests belonging to different tenants, to the same microservice [16]. On the other hand, for traditional datacenter applications, multi-tenancy is handled at a very coarse granularity where entire applications belonging to different users are co-scheduled [27, 31, 44, 45, 48]. These observations clearly point to the need for a paradigm shift in the design of runtime systems that can enable and drive multi-tenant execution where different jobs share common microservices in a microservice design framework.

Towards rethinking runtime systems that drive multi-tenancy in microservice design frameworks, we seek to identify and exploit key new opportunities that exist, in this context. **First, the ability to accurately predict the time each request spends at a microservice even prior to its execution opens up a key opportunity towards performing safe consolidations without violating SLAs.** This, when exploited judiciously, could enable the sharing of microservices that are employed across multiple jobs, achieving high throughput, while still meeting SLAs. **Second, the variability existing in SLAs when multiple latency sensitive jobs are consolidated generates a lot of request level execution slack that can be distributed across other requests.** In other words, consolidated execution is avoided for requests with low execution slack and vice versa. These scenarios create new opportunities in the microservice execution framework to achieve high throughput by consolidating the execution of multiple latency sensitive jobs, while still achieving user-specific SLAs, through fine-grained task management.

3 Analysis of Microservices

This section investigates the performance characteristics of emerging AI and ML services utilizing the pipelined microservices. Using that, we develop a methodology that can accurately estimate completion time for any given request at each microservice stage prior to its execution. This information becomes beneficial towards safely enabling fine-grained request consolidation when microservices are shared among different applications under varying latency constraints.

3.1 Performance of Microservices

In this section, we analyze three critical factors that determine the execution time of a request at each microservice stage: **(1) Sharing degree (2) Input size (3) Queuing delay**. For this analysis, we select a microservice that performs `image classification` (IMC) which is a part of the catalog of microservices offered by AWS Step Functions [39].

(1) Sharing degree. Sharing degree defines the granularity at which requests belonging to different jobs (or applications) are batched together for execution. A sharing degree of one means that the microservice processes only one request at a time. This situation arises where a microservice instance executing a job restricts sharing its resources simultaneously for requests belonging to other jobs. Requests under this scheme can achieve low latency at the cost of low resource utilization. On the other hand, a sharing degree of thirty indicates that the microservice merges thirty requests into a single batch to process the requests belonging different jobs simultaneously. Increasing the sharing degree has demonstrated to increase the occupancy of the underlying computing platform (especially for GPUs) [16]. However, it has a direct impact on the latency of the executing requests as the first request arriving at the microservice would end up waiting until the arrival of the 30th request when the sharing degree is 30.

Figures 3a and 3b illustrate the impact of sharing degree on latency and throughput. The inputs that we have used for studying this effect is a set of images with dimension 128x128. The horizontal axes on both figure 3a and 3b represent the sharing degree. The vertical axis in figure 3a and 3b represents latency in milliseconds and throughput in requests per second respectively. From figures 3a and 3b, we can clearly see that the sharing degree improves throughput. However, it affects the latency of execution of individual requests as well.

(2) Input size. Second, we observe changes in the execution time of a request by varying its input size. As the input size increases, additional amounts of computation would be performed by the microservices. Hence, input sizes play a key role in determining the execution time of requests. To study this using the image classification (IMC) microservice, we obtain request execution times for different input sizes of images from 64x64 to 256x256. The sharing degree is kept constant in this experiment. Figure 3c illustrates the findings of our experiment. We observe that as input sizes increase, execution time of requests also increase. We also observed similar performance trends for other microservice types.

(3) Queuing delay. Queuing delay is the last factor that affects execution time of requests. This is experienced by requests waiting on previously dispatched requests to be completed. From our analysis, we observe that there is a linear relationship between the execution time of a request

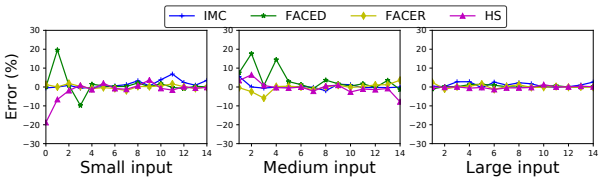


Figure 4. Error(%) in predicting ETC for different input sizes with increase in the sharing degree (x-axis)

its sharing degree and input size respectively. Queuing delay can be easily calculated at runtime using the execution sequences of requests, the estimated execution time of individual requests and its preceding requests.

From these observations, we conclude that there is an opportunity to build a highly accurate performance model for each microservice that our execution framework can leverage to enable sharing of resources across jobs. Further, we also provide capabilities that can control the magnitude of sharing at every microservice instance. These attributes can be utilized simultaneously for preventing SLA violations due to microservice sharing while optimizing for datacenter throughput.

3.2 Execution Time Estimation Model

Accurately estimating the execution time of a request at each microservice stage is crucial as it drives the entire microservice execution framework. Towards achieving this, we try to build a model that calculates the estimated time of completion (ETC) for a request at each of its microservice stages. The ETC of a request is a function of its compute time on the microservice and its queuing time (time spent waiting for the completion of requests that are scheduled to be executed before the current request).

$$ETC = T_{queuing} + T_{compute} \quad (1)$$

We use a linear regression model to determine the $T_{compute}$ of a request, for each microservice type and the input size, as a function of the sharing degree.

$$Y = a + bX \quad (2)$$

where X is the sharing degree (batch size) which is an independent variable and Y is the dependent variable that we try to predict, the completion time of a request. b and a are the slope and intercepts of the regression equation. $T_{queuing}$ is determined as the sum of the execution times of the previous requests that need to be completed before the current request can be executed on the microservice which can directly be determined at runtime. Each model obtained is specific to a single input size. Hence, we design a system where we have a model for every specific input size that can predict ETC for varying batch sizes and queuing delays.

Data normalization. A commonly followed approach in machine learning is to normalize data before performing linear regression so as to achieve high accuracy. Towards this objective, we rescale the raw input data present in both dimensions in the range of [0, 1], normalizing with respect to the min and max, as in the equation below.

$$x' = \frac{x - min(x)}{max(x) - min(x)} \quad (3)$$

We trained our model for sharing degrees following powers of two to create a predictor corresponding to every microservice and input size pair. We cross validated our trained model by subsequently creating test beds and comparing the actual values with the estimated time of completion by our model. Figure 4 shows the error rate that exists in predicting the completion time, given a sharing degree for different input sizes. For the image based microservices, the input sizes utilized are images of dimensions 64, 128 and 256 for small, medium and large inputs, respectively. These are standardized inputs from publicly available datasets whose details are enumerated in Table 1. As can be clearly observed from the graph, the error in predicting the completion time from our model is around 4% on average. This remains consistent across other microservices too whose plots are not shown in the figure to avoid obscurity.

The estimated time of completion (ETC) obtained from our regression models is used to drive decisions on how to distribute requests belonging to different users across microservice instances. However, satisfying application-specific SLAs becomes mandatory under such circumstances. For this purpose, we seek to exploit the variability in the SLAs of individual requests and the resulting slack towards building our request scheduling policy. Later in section 4.2 and 4.3, we describe in detail the methodology by which we compute and utilize slack to undertake optimal request distribution policies.

The ETC prediction model that we have developed is specific towards microservice types whose execution times can be predicted prior to its execution. Based on our observations, applications belonging to the AI and ML space exhibit such execution characteristics and fit well towards being part of microservice execution frameworks hosted at Serverless Computing Infrastructures. However, there exist certain microservice types whose execution times are highly unpredictable. For instance, an SQL range query's execution time and output is dependent both on the input query type and the data which it is querying. Such microservice types cannot be handled by our model. We discuss this at much more detail in Section 5.

4 GrandSLAm Design

This section presents the design and architecture of GrandSLAm, our proposed runtime system for moderating request distribution at micro-service execution frameworks. The goal

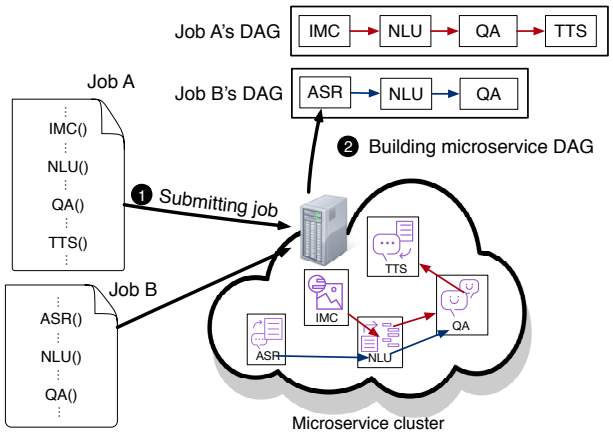


Figure 5. Extracting used microservices from given jobs in the microservice cluster

of GrandSLAm is to enable high throughput at microservice instances without violating application specific SLAs. GrandSLAm leverages the execution time prediction models to estimate request completion times. Along with this GrandSLAm utilizes application/job specific SLAs, to determine the execution slack of different jobs' requests at each microservice stage. We then exploit this slack information for efficiently sharing microservices amongst users to maximize throughput while meeting individual users' Service Level Agreements (SLAs).

4.1 Building Microservice Directed Acyclic Graph

The first step in GrandSlam's execution flow is to identify the pipeline of microservices present in each job. For this purpose, our system takes the user's job written in a high-level language such as Python, Scala, etc. as an input (❶ in Figure 5) and converts it into a directed acyclic graph (DAG) of microservices (❷ in Figure 5). Here, each vertex represents a microservice and each edge represents communication between two microservices (e.g., RPC call). Such DAG based execution models have been widely adopted in distributed systems frameworks like Apache Spark [50], Apache Storm [21], TensorFlow [1],etc. Building a microservice DAG is an offline step that needs to be performed once before GrandSLAm's runtime system starts distributing requests across microservice instances.

4.2 Calculating Microservice Stage Slack

The end-to-end latency of a request is a culmination of the completion time of the request at each microservice stage. Therefore, to design a runtime mechanism that provides end-to-end latency guarantees for requests, we take a disaggregated approach. We calculate the partial deadlines at each microservice stage which every request needs to meet at so that end-to-end latency targets are not violated. We define

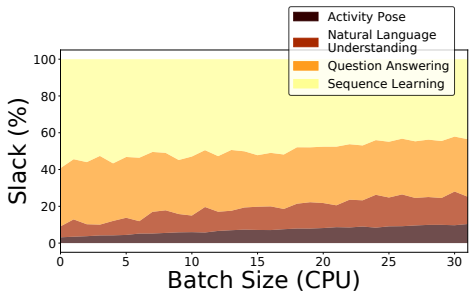


Figure 6. Microservice stage slack corresponding to different microservices present in Pose Estimation for Sign Language application

this as **microservice stage slack**. In other words, microservice stage slack is defined as the maximum amount of time a request can spend at a particular microservice stage. Stage slacks are allocated offline after building the microservice DAG, before the start of the GrandSLAm runtime system.

Mathematically slack at every stage is determined by calculating the proportion of end-to-end latency that a request can utilize at each particular microservice stage.

$$slack_m = \frac{L_m}{L_a + L_b \cdots + L_m + \ldots} \times SLA \tag{4}$$

where L_m is the latency of job at stage m and L_a, $L_b \ldots$ are the latency of the same job at the other stages $a, b \ldots$ respectively. Figure 6 illustrates the proportion of time that should be allocated at each microservice stage for varying batch sizes, for a real world application called Pose Estimation for Sign Language. We can clearly see from Figure 6 that the percentage of time a request would take to complete the Sequence Learning stage is much higher than the percentage of time the same request would take to complete the Activity Pose stage. Using this observation, requests are allocated stage level execution slacks proportionally.

4.3 Dynamic Batching with Request Reordering

GrandSLAm's final step is an online step orchestrating requests at each microservice stage based on two main objective functions (i) meeting end-to-end latency (ii) maximizing throughput. For this purpose, GrandSLAm tries to execute every request that is queued up at a microservice stage in a manner at which it simultaneously maximizes the sharing degree while meeting end-to-end latency guarantees. In this regard, GrandSLAm undertakes two key optimizations: ❶ Request reordering and ❷ Dynamic batching as depicted in Figure 7. GrandSLAm through these optimizations tries to maximize throughput. However, it keeps a check on the latency of the executing job by comparing slack possessed by each request (calculated offline as described at 4.2) with its execution time estimates (obtained from the model described at Section 3.2).

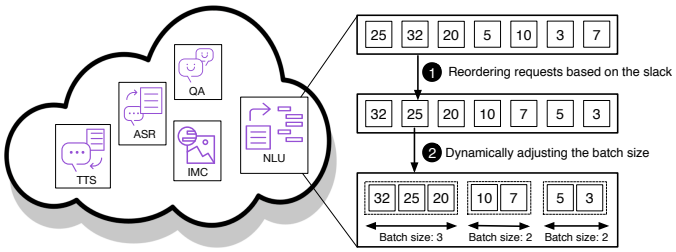


Figure 7. Request reordering and dynamic batching mechanism

Request reordering. Slack based request reordering is performed at each microservice instance by our runtime system. The primary objective of our request reordering mechanism is to prioritize the execution of requests with lower slack as they possess much tighter completion deadlines. Hence, our GrandSLAm runtime system reorders requests at runtime that promotes requests with lower slack to the head of the execution queue. The request reordering mechanism in Figure 7 illustrates this with an example. Each rectangle is a request present in the microservice execution and the number in each rectangle illustrates its corresponding slack value. On the left, it shows the status before reordering, and on the middle, it shows the status after reordering.

Dynamic batching. At each microservice stage, once the requests have been reordered using slack, we identify the largest sharing degree (actual batch size during execution) that can be employed such that each request's execution time is within the allocated microservice stage slack. Such a safe identification of the largest sharing degree is done by comparing the allocated slack obtained by the process described in Section 4.2 with the execution time estimation model described in Section 3.2.

Algorithm 1 summarizes the dynamic batching approach that we employ. The input to the algorithm is a queue of requests sorted by their respective slack values. Starting from the request possessing the lowest slack value we traverse through the queue increasing the batch size. We perform this until increasing batch size violates the sub-stage SLA of individual requests present in the queue. We repeat the request reordering and dynamic batching process continuously as new incoming requests arrive from time to time. Figure 7 shows how the dynamic batching is used in our system from the middle part to the right part.

4.4 Slack Forwarding

While performing slack based request scheduling in multi-stage applications, we observed a common scenario. There is always some leftover slack that remains unused for many requests. For instance, at the first stage if the best ETC value provided for a request is 100ms and the slack allocated for that stage is 135ms, there is 35ms (135ms - 100ms) leftover slack. We reutilize this remaining slack, by performing slack forwarding, wherein we carry forward the unused slack on

Algorithm 1 Dynamic batching algorithm

```
1:  procedure DynBatch(Q)                          ▷ Queue of requests
2:      startIdx = 0
3:      Slack_q = 0
4:      executed = 0
5:      len = length(Q)
6:      while executed ≤ QSize do                  ▷ All are not batched
7:          window = 0
8:          partQ = Q[startidx : length]
9:          window = getMaxBatchSizeUnderSLA(partQ, Slack_q)
10:         startIdx = startIdx + window
11:         Slack_q = Slack_q + latency
12:         executed = executed + window
13:     end while
14: end procedure
```

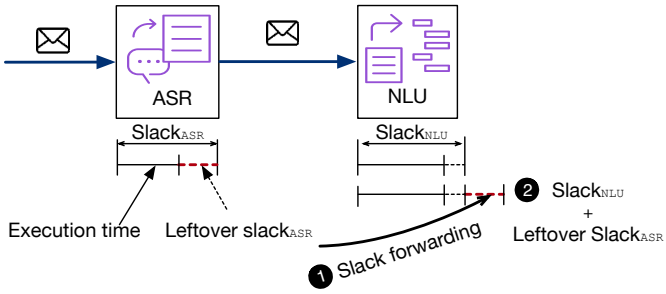


Figure 8. Forwarding unused slack in the ASR stage to the NLU stage

to the subsequent microservice stages. Figure 8 exemplifies the case where the unused slack in the ASR stage can be forwarded into the next NLU microservice stage. This has increased the overall request slack in the later stages of execution in a multi-stage application enabling higher sharing degrees.

5 Discussion

Our approach requires an accurate estimation of the execution time at each microservice stage. For this purpose, it becomes essential to determine the factors affecting the execution time of microservices. This motivated us to develop an execution time estimation (ETC) model based on a set of factors based on the application space we have considered. In this study, we analyzed the performance characteristics of AI and ML related microservices as these applications were well suited to be hosted on the microservice architecture. In this context, we observed two distinct characteristics in the AI and ML space. First, batching multiple requests into a single large one is widely used in these microservices to improve the resource utilization of the computing devices. For this purpose, these microservices performs preprocessing of inputs (e.g., resizing images in image classifications, splitting voice inputs in speech recognition, chunking words in natural language processing) to fit in a single batch for simultaneous execution. Second, many of the AI applications exhibit the pipelined execution of microservices. Image recognition, an application from AWS Step Functions [39] is one such example. Such simple linear pipelines make it much easier to design slack-based optimizations introduced in Section 4.

Limitations. However, we anticipate that our methodology cannot be applied directly to microservice types other than AI and ML space. For example, a simple model that we have proposed is not sufficient for other types of microservices which do not batch queries belonging to different applications. For example, the execution time of the microservices executing SQL range queries, will be sensitive on both the input query and output results. In other words, similar queries executed on different datasets might possess different execution times. In such circumstances, it requires a much more detailed analysis and investigation on application types for building much more sophisticated models. In addition to that, the complex microservice topologies such as general graphs and conditional execution have not been considered in this study. It is challenging for GrandSLAm in its existing form to calculate slacks in cases where different requests take different paths at runtime or need to perform a few microservices in parallel. These are some of the limitations of GrandSLAm which we plan to investigate in the near future.

6 Evaluation

In this section, we evaluate GrandSLAm's policy and also demonstrate its effectiveness in meeting service level agreements (SLAs), while simultaneously achieving high throughput in datacenters that house microservices.

6.1 Experimental Environments

Infrastructure. We evaluate GrandSLAm on a testbed consisting of 100 docker containers. Each container has a single 2.4 GHz CPU core, 2GB of RAM and runs Ubuntu 16.10. GrandSLAm is evaluated on both CPU as well as GPU platforms as enumerated in Table 2. Today's datacenters house different kinds of accelerators improving the performance of AI and ML applications [16, 17, 34, 36, 38]. We setup a topology of services and processes according to that of **IBM Bluemix** [13]. In other words, each microservice executes on containerized execution environments. We use docker containers for this purpose.

Microservice types. Table 1 shows the list of microservices that we have utilized in our experiments. POS, CHK, and NER microservices utilize the kernels from Djinn&Tonic [16] suite which in turn uses SENNA [9]. Similarly, ASR microservice utilizes kernels from Djinn& Tonic suite [16], which in turn uses Kaldi [37]. IMC, FACED, FACER, AP, HS, QA, and SL microservices are implemented using **TensorFlow framework version 1.0** [1].

Type	Application	Input Sizes	Output	Network	Type	Layers	Parameters
Image Services	Image Classification (IMC)	64X64, 128X128 and 256 X 256 images	Probability of an object	Alexnet	CNN	8	15M
	Face Detection (FACED)		Facial Key Points	Xception	CNN	9	58K
	Facial Recognition (FACER)		Probability of a person	VGGNet	CNN	14	40M
	Human Activity Pose (AP)		Probability of a pose	deeppose	CNN	8	40M
	Human Segmentation (HS)		Presence of a body part	VGG16	CNN	16	138M
Speech Services	Speech Recognition (ASR)	52.3KB, 170.2KB audio	Raw text	NNet3	DNN	13	30M
	Text to Speech (TTS)		Audio output	WaveNet	DNN	15	12M
Text Services	Part-of-Speech Tagging (POS)	text containing 4-70 words per sentence	WordâĂŹs part of speech eg. Noun	SENNA	DNN	3	180K
	Word Chuncking (CHK)		Label Words as begin chunk etc.	SENNA	DNN	3	180K
	Name Entity Recognition (NER)		Labels words	SENNA	DNN	3	180K
	Question Answering (QA)		Answer for question	MemNN	RNN	2	43K
	Sequence Learning (SL)		Translated text	seq2seq	RNN	3	3072
General Purpose Services	NoSQL Database (NoSQL)	Directory input	Output of Query	N/A	N/A	N/A	N/A
	Web Socket Programmig (WS)	Text, image	Data communication	N/A	N/A	N/A	N/A

Table 1. Summary of microservices and their functionality

CPU/GPU config	Microarchitecture
Intel Xeon E5-2630 @2.4 GHz	Sandy Bridge-EP
Intel Xeon E3-1420 @3.7 GHz	Haswell
Nvidia GTX Titan X	Maxwell
GeForce GTX 1080	Pascal

Table 2. Experimental platforms

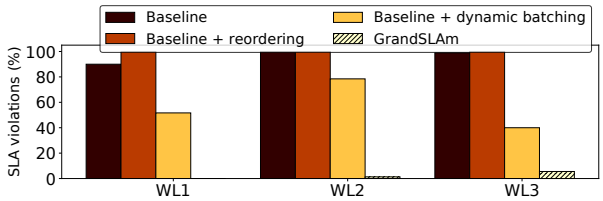
Application	Description	Pipelined microservices
IPA-Query	Provides answers to queries that are given as input through voice.	ASR→NLP→QA
IMG-Query	Generates natural language descriptions of the images as output.	IMG→NLP→QA
POSE-Sign	Analyzes interrogative images and provides answers.	AP→NLP→QA→SL
FACE-Security	Scans images to detect the presence of identified humans.	FACED→FACER
DETECT-Fatigue	Detects in real time the onset of sleep in fatigued drivers.	HS→AP→FACED→FACER
Translation	Performs language translation.	SL QA NoSQL

Table 3. Applications used in evaluation

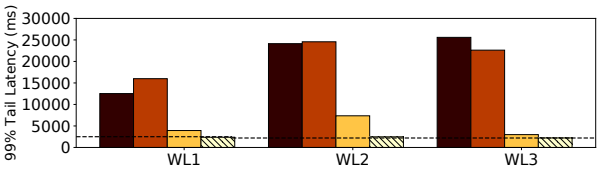
	Applications	Shared microservices
WL1	IMG-Query, FACE-Security, DETECT-Fatigue, POSE-Sign	QA, FACED, FACER, AP
WL2	IPA-Query, POSE-Sign, Translation	NLU, QA
WL3	I/O-IPA-Query, I/O-Sign, I/O-Translation	NLU, NoSQL

Table 4. Workload scenarios

Load generator/Input. To evaluate the effectiveness of GrandSLAm, we design a load generator that submits user requests following a Poisson distribution that is widely used to mimic cloud workloads [33]. The effect of performance degradation at multi-tenant execution scenarios is luminous extensively at servers handling high load. Hence, our experiments are evaluated at scenarios in datacenters where the load is high. Such a distribution has been used by several prior works on multi-stage applications [42, 47, 49]. The SLA that we use for each application is obtained and calculated from the methodology proposed by PowerChief [49]. Table 4 shows the workload table and the microservices that are shared when they are executed together. For each microservice request we have evaluated our methodology using inputs that correspond to data that is available from open source datasets.



(a) Percentage of requests that violate SLA



(b) 99th Percentile tail latency of each application

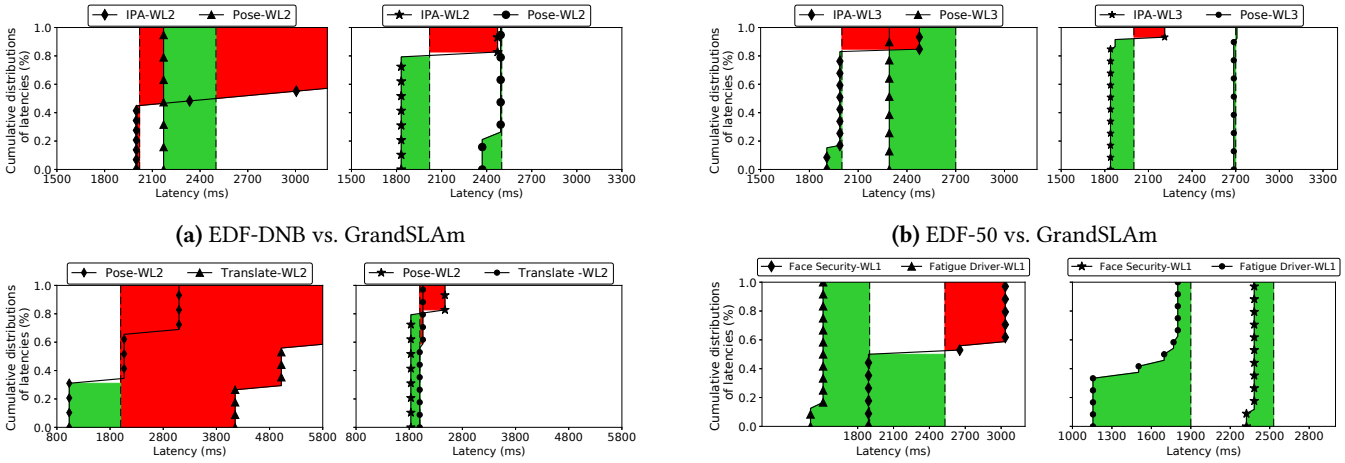
Figure 9. Comparing the effect of different components present in GrandSLAm's policy

6.2 Achieving Service Level Agreements (SLAs)

First, we evaluate the effectiveness of GrandSLAm in achieving Service Level Agreements (SLAs) for the workload scenarios enumerated in Table 4. For this purpose, we introduce reordering and batching incrementally over the baseline system and try to study its effects on the percentage of SLA violations.

6.2.1 Reducing SLA Violations

For this experiment, we deployed a docker container instance for each microservice type. Communication across microservice instances within the cluster happens through web sockets. Under this experimental setup, we first obtain the percentage of requests violating SLA under a **baseline scheme** which executes requests (i)in a first-in-first-out (FIFO) fashion (ii)without sharing the microservices. Subsequently, we introduce a request re-ordering scheme that executes requests in an Earliest Deadline First (EDF) fashion to compare it with the baseline system. Similarly, we also execute requests in a situation where requests share microservice instances(using query batching) to see how it



(a) EDF-DNB vs. GrandSLAm

(b) EDF-50 vs. GrandSLAm

(c) ED-DNB vs. GrandSLAm

(d) ED-30 vs. GrandSLAm

Figure 10. Comparing the cumulative distribution function of latencies for prior approaches and GrandSLAm.

improves performance. Lastly, we compare GrandSLAm with these schemes to illustrate its effectiveness. Our experiment keeps the input load constant at fixed Requests per Second (RPS) while comparing each policy.

Figure 9 shows the results of this experiment. From Figure 9a, we can clearly see that for a given workload, almost all of the requests violate SLAs under the baseline and reordering policies. However, the effect is much amortized when requests are grouped together in batches. This is because batching can improve the overall latency of a multitude of requests collectively [16]. This is clearly evident from the percentage of requests violated under baseline+dynamic batching policy. GrandSLAm utilizes best of both the policies where it ends up having a low percentage of requests that violate SLAs.

6.3 Comparing with Prior Techniques

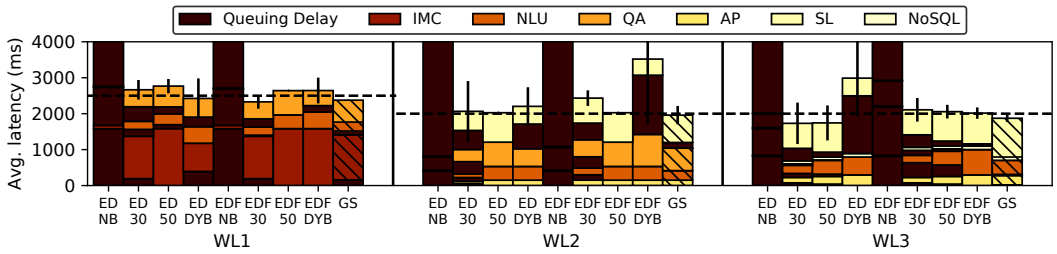
Prior approaches which try to solve this problem are categorized based on their respective (i) batching policies for aggregating requests and (ii) slack calculation policies for reordering requests. Most relevant work use a no-batching policy where they do not batch multiple requests. Djinn&Tonic [16] utilizes a static batching policy where they used a fixed batch size for all applications. However, we propose a dynamic batching technique which varies the batch size based on the slack available for each request. Again, with respect to slack calculation policy, prior approaches [21, 50] utilize an equal division slack allocation (ED) policy which equally divides slack across individual microservice stages. Certain other approaches utilize a first-in-first-out policy while most approaches utilize earliest deadline first (EDF) slack allocation policy [42, 47]. However, we propose a slack calculation policy which allocates slack taking into account the intrinsic variation present in the execution time of different computational stages. This is explained in Section 4.2.

We derive 4 baselines on equal division policy. `ED-NB` (equal division no batch) disables batching, `ED-30` and `ED-50` statically fix batch size to 30 and 50 respectively, and `ED-DNB` (equal division dynamic batch) utilizes the dynamic batching approach proposed by GrandSLAm along with the `ED` policy. We also derive 4 baselines on using earliest deadline first policy: `EDF-NB`, `EDF-30`, `EDF-50` and `EDF-DNB`, respectively. GrandSLAm's policy is abbreviated as `GS` in our graphs.

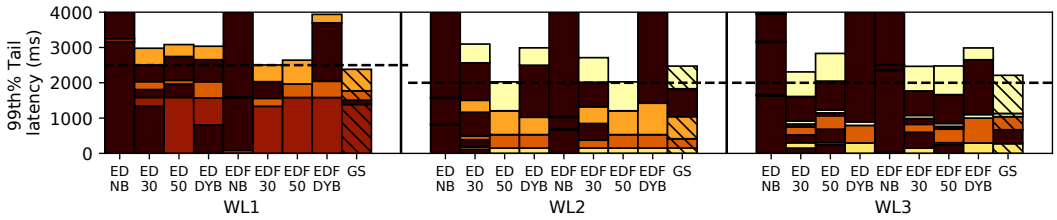
6.3.1 Reordering Requests based on Slack

In this subsection, we quantify the effectiveness of GrandSLAm's slack calculation and reordering policy by comparing it with `ED` and `EDF`. We illustrate this using the cumulative distribution function (CDF) of latencies, as shown in Figure 10. We have used the same experimental setup where the configuration of the input load and the number of microservice instances remains constant.

Figures 10a, 10b, 10c and 10d compare the cumulative distribution function (CDF) of the policies `EDF-DYN`, `EDF-50`, `ED-DYN`, and `ED-30`, respectively with GrandSLAm. The horizontal axis denotes time. The vertical axis denotes the CDF of the percentage of requests executed at a particular time. The dashed lines correspond to the target SLAs that individual applications are subjected to meet. For each figure, the graph in the left portrays the CDF of the baseline techniques (`EDF-DYN`, `EDF-50`, `ED-DYN`, and `ED-30`) and the graph in the right portrays the CDF of GrandSLAm. The green shaded portion illustrates the leftover slack at the final stage when requests execute before the deadline. The red shaded portion illustrates slack violation when requests execute after the deadline has passed. In an ideal case, both green and red portions should be minimized. In other words, requests should



(a) Average latency



(b) 99th Percentile tail latency

Figure 11. Comparing the latency of workloads under different policies. GrandSLAm has the lowest average and tail latency.

be reordered and batched in such a way that it neither passes the deadline nor executes way ahead of the deadline. Executing way ahead of the deadline restricts requests with lower slack to stall creating a situation where other requests end up violating SLAs. In an ideal situation, slack remaining should be transferred to the requests who are about to violate slack. From these graphs, we draw the following conclusion. As shown in Figure 10a, 10b, 10c and 10d requests reordering policies proposed by prior literature creates a situation where a few requests execute much before the expected deadline while other requests end up violating the SLAs.

Figure 10a and 10b compare EDF with GrandSLAm. EDF's slack allocation policy for each request is agnostic to the intrinsic variation present in the microservice execution stages within an application. Hence, in many instances, it underestimates execution times of requests and performs aggressive batching. As a result of this, some requests complete their execution well ahead of the latency targets while other requests end up violating SLAs. GrandSLAm, on the other hand, avoids this situation by allocating slack that is proportional to the time that would be taken at each stage. GrandSLAm performs judicious batching while limiting aggressive batching by introducing sub-stage SLAs. This is clearly illustrated in Figure 10a. Pose and IPA are two applications present in WL2. Under EDF's policy, we see that the requests corresponding to the Pose application complete well ahead of time (as shown in the green patch). However, a substantial number of requests corresponding to the IPA violate SLAs(as shown in the red patch). GrandSLAm, on the other hand, carefully reallocates slack among applications. Hence, the execution of requests with abundant slack is stalled until just before the deadline thereby allowing requests with less amount of slack to be executed, preventing them from violating SLAs. The can clearly be seen in Figure 10a as the amount of green and red patches are much lesser for GrandSLAm. A similar phenomenon can be witnessed for EDF's static batching policy with batch size 50 in Figure 10b.

Figure 10c and 10d compare ED with GrandSLAm. The major drawback of the ED technique lies in its inability to gauge the slack that should be allocated in each stage. This is clearly illustrated in Figure 10c and 10d. In many cases, it wrongly allocates more slack to requests that do not require it, while depriving other requests that actually need slack. This introduces additional queuing time, thereby violating the SLA for a substantial amount of requests. This could be avoided if slack is being distributed judiciously across requests. GrandSLAm is cognizant of this need and hence, predicts the appropriate amount of compute time required for each stage and allocates slack proportionally.

6.3.2 Dynamic Batching for Latency Reduction

In order to study the effects of dynamic batching, we compare GrandSLAm with all our baseline policies. Figure 11 and 12 illustrate the results of this experiment. In Each stacked bar in Figure 11a and 11b represents the average latencies and the tail latencies of the applications respectively. The policies in each figure are ordered starting from ED-NB followed by ED-30, ED-50, ED-DYN, EDF-NB, EDF-30, EDF-50, EDF-DYN concluding with GrandSLAm as GS respectively. GrandSLAm is distinctively distinguished from other bars by hatching it with slanting lines. The color in the stacked graph corresponds to either queuing latency experienced at any stage or the compute latency at individual microservice stages. The different components of this plot are stacked

breaking the end-to-end latency as queuing latency or compute stage delay over time (which is why there is a queuing latency stack after each stage). As can be seen in Figure 11a, GrandSLAm achieves the lowest latency across all policies. GrandSLAm is able to meet the required SLA for almost every request, as compared to prior policies that violate SLAs for several of these requests. We draw the following insights into why prior policies are ineffective in meeting SLAs.

No batching techniques. The latency of requests is completely dominated by the queueing latency when employing techniques that don't perform batching, namely ED-NB and EDF-NB. Hench, such policies are undesirable.

Static batching techniques. In view of the clear disadvantage when requests are not batched, statically batching them is one of the simplest policies that can be employed to improve throughput. However, latencies and SLAs could be compromised if they are not batched judiciously.

Assigning a large fixed batch size for execution can critically violate the latency of many requests within that particular batch. Let us take WL1 for example. From Figure 11a and 11b we see that employing a fixed batch size (batch size 50) under EDF policy violates SLA only by a small proportion. However, it violates the SLA for most requests present in the workload. This can be seen in Figure 12 where the percentage of violations for WL1 under ED-50 goes up to 60%. This is caused because of using a large batch size resulting in a situation where every request ends up violating the SLA especially at the last stage of the application. This is because a fixed batch size is not aware of the latencies and slack of requests that are executing at a point in time. This is an unfavorable outcome especially for applications that require strict latency targets.

To remedy this, employing smaller batch sizes could be viewed as a favorable solution. However, smaller batch sizes can be conservative, thereby not being able to exploit the potential opportunities where aggressive batching can increase throughput while still meeting the latency constraints. Furthermore, small batch sizes could also cause excessive queuing. Specifically, when requests are grouped with small batch sizes, the first few batches might have low queuing delays. However subsequent batches of requests would end up waiting for a substantial period of time for the execution of prior batches of requests to complete, thereby affecting the end-to-end latency. This increase in queueing latency at the later stages can be clearly seen in situations created by WL2 (from figure 11a and 11b) where policies ED-30 and EDF-30 violates SLAs both in terms of average latencies as well as tail latencies. Additionally, many requests also violate SLAs as queuing becomes a huge problem due to large batch sizes. This can be seen in figure 12. These observations strongly motivate a dynamic batching policy where batch sizes are determined online, during runtime, depending upon each application's latency constraints.

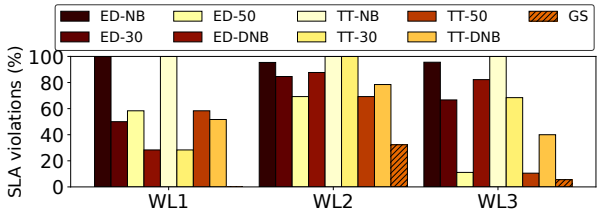


Figure 12. Percentage of requests violating SLAs under different schemes

Dynamic batching. Equal Division dynamic batching, Earliest Deadline First dynamic batching and Grand Slam determines appropriate batch sizes during runtime. The difference between these three policies is the way by which they compute slack. Once slack is computed, the largest batch size which accommodates all the requests without violating its slack is obtained during runtime. For Equal Division dynamic batching, slack for each request is a fair share from the SLA for each stage in the end-to-end pipeline. For instance, for an application consisting of 3 stages, each request of that application is estimated to have a slack of 33% of the SLA at each stage. Earliest Deadline first approach, however, undertakes a greedy approach wherein the slack for each request of an application at each stage is the remaining time the request possesses before it would end up violating the SLA. GrandSLAm is unique and distinct from all these mechanisms. We adopt the methodology elaborated in Section 4.2 that is cognizant of the volume of computation each individual stage performs.

In Figures 11a and 11b we clearly see that both Equal Division dynamic batching and Earliest Deadline First dynamic batching perform poorly. This is due to the following reasons. First, the policy that Earliest Deadline First (EDF) utilizes to determine the appropriate batch size for a set of requests is a greedy policy. EDF dynamically selects batch sizes for the requests aggressively until there is remaining slack. Although this can be beneficial for traditional datacenter applications where execution can only be thought of as single stage and monolithic, such an approach performs poorly at microservice execution framework that possesses multistage execution pipelines. This is due to the fact that when requests reach the final stages of execution, they have a limited amount of slack, which in turn restricts the amount of batching possible to avoid potential SLA violations due to excessive batching. Such a policy has two key downsides, First, it increases the queuing time for subsequent requests thereby increasing the of those requests. This has a negative impact especially on the tail latency of applications as shown in figure 11b. Second, it becomes difficult to identify the exact individual stage that was the causing this bottleneck. As a result, the command center will perform non-optimal remediation where unwanted instances would be scaled up leading to high resource utilization. This is experimentally validated

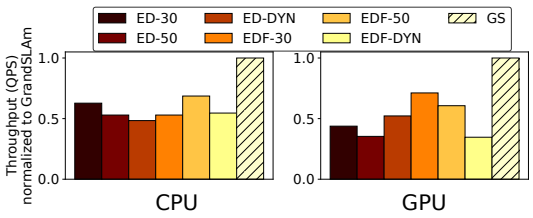


Figure 13. Throughput gains from GrandSLAm

in section 6.4.2. Third, equal division dynamic batching introduces a fair share of sub-stage SLA for each stage. This can restrict microservices from batching aggressively at a single stage. It can also identify the exact microservice instance that was responsible for end-to-end SLA violation. Such a policy, on the one hand, can address the high tail latency problem that exists in the Earliest Deadline First's aggressive and greedy dynamic batching approach. However, on the other hand, it neglects the fact that the computation time at each stage is very different. Hence, in many scenarios, it does not exploit the full benefits of batching for stages that have high slack. For example, during the final stage in WL2 shown in figure 11a and figure 11b, if all the requests have been batched, the percentage of requests that would have violated slack would be much lower. However, the equal division policy cannot exploit this opportunity resulting in an increased latency of requests.

GrandSLAm. Our technique, on the other utilizes a hybrid approach by exploiting the advantages of dynamic batching as well as enabling sub stage cut off slacks. GrandSLAm utilizes a weighted sub-stage SLA slack based on the computational requirements of each stage and an online dynamic batching technique. As a result, GrandSLAm is able to outperform all prior approaches and achieve a much lower average and tail latency as shown in figures 11a and 11b.

6.4 GrandSLAm Performance

In this section, we evaluate GrandSLAm's capability in increasing datacenter throughput and server utilization, while guaranteeing Service Level Agreements (SLAs) for the workload scenarios enumerated in Table 4.

6.4.1 Throughput Increase

In this section, we demonstrate the throughput benefits of GrandSLAm, as compared to the state-of-the-art techniques at scale-out environments. We compare the different execution policies by constructing a real-time simulational experimental setup consisting of a 1000 node CPU and GPU enabled cluster. As executing AI applications in accelerator platforms is becoming more common, we try to evaluate our technique at both CPU and GPU platforms. For GPU based experiments the executing workloads do not utilize the CPU and are executed only in the GPU device and vice versa. Additionally, to mimic scale out execution scenarios, we collect performance telemetry of workload scenarios for multiple execution runs. We then extrapolate the performance telemetry to obtain data nearly equivalent to the amount of data being collected at large scale datacenter. On top of that, we build a simulation infrastructure that mimics GrandSLAm's execution model at a larger scale. We also fix our application specific SLA, instance count and the server configuration across experimental runs. We ensure that every request executing across the end-to-end pipeline meets the latency constraints. Under such situations, we observe the throughput gains corresponding to each execution policy.

Figure 13 illustrates the throughput gains of GrandSLAm compared to state of the art execution policies. Each bar represents the average number of Requests executed per Second (RPS) across all the applications and workload scenarios enumerated in Table 4, normalized to the average QPS of GrandSLAm. We normalize with respect to GrandSLAm since the best prior technique is different for the CPU and GPU systems. We clearly see that GrandSLAm outperforms other execution policies. The graph on the left is the average throughput for executing the workloads on a CPU cluster while the graph on the right illustrates the results of the same experiment on a GPU platform. An interesting observation consistent across both CPU and accelerator platforms is that the static batching techniques consistently outperform the dynamic batching techniques. This is because, dynamic batching, for instance, in the context of time trader, aggressively batches requests initially. However, requests get stalled during the terminal stages resulting in decreased throughput. On the contrary, equal division misjudges the proportion of slack that is to be allocated. As a result, the policy restricts aggressive batching during scenarios where latency does not take a hit. This results in low throughput. On an average we obtain up to 3× performance on the GPU platform and around 2.2× performance on the CPU server cluster, over the best prior mechanism.

6.4.2 Reduced Overheads

In this section, we illustrate the decrease in the number of microservice instances when employing GrandSLAm's execution policy. Under fixed latency and throughput constraints, we try to obtain the number of microservice instances of each type that is required for executing the workloads enumerated in Table 4 in a scale-out fashion similar to section 6.4.1.

Figure 14 compares the instance count for GrandSLAm and prior works. The top graph corresponds to CPU performance while the bottom graph corresponds to GPU performance. We can see that GrandSLAm reduces instance count significantly on both the CPU and GPU platforms. Additionally, GrandSLAm's instance count reduction is higher on the GPU platform. This is intuitive as GPUs are devices that are optimized to provide high throughput. Overall, we conclude that GrandSLAm is able to effectively meet SLAs while achieving high throughput at low instance counts.

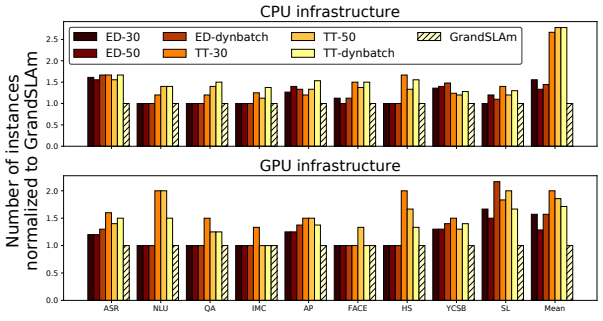


Figure 14. Decrease in number of servers due to GrandSLAm

7 Related Work

Prior literature on guaranteeing response latency falls into two primary categories: Improving QoS without violating latency constraints and managing SLAs in multi-stage applications.

7.1 Improving QoS without Latency Violation

Prior work on addressing response latency variation and providing quality of service (QoS) guarantees have primarily been in the context of traditional datacenters [10, 31, 35, 48]. Bubble-Up [31] and Bubble-Flux [48] quantify contention for last level cache and memory bandwidth towards enabling co-location of a latency critical application alongside batch applications. However, these techniques prioritize the latency critical user-facing application and end up significantly hurting the performance of the co-running batch applications. Paragon [10] and Whare-Map [30] utilize runtime systems using machine learning techniques like collaborative filtering and sensitivity analysis towards identifying the right amount of resources required for guaranteeing QoS in heterogeneous datacenters. However, these techniques are designed for traditional datacenter applications like memcached, web search, etc. There is some prior literature that attempts to estimate performance at co-located situations in accelerator environments [2, 7, 8, 23, 28, 43]. Baymax [8] predicts the behavior of tasks executing in a GPU accelerator context. Prophet [7] models the interference across accelerator resources in co-located execution scenarios. However, neither of these techniques caters to the needs of a microservice execution framework, as they do not tackle the challenge of providing solutions for guaranteeing latency for applications containing multiple stages.

7.2 Managing SLAs in Multi-Stage Applications

Recent prior studies have identified the advantages of applications that are composed of multiple stages, especially its ease of deployment [12, 18, 19, 22, 24, 25, 38, 42, 46]. Under such scenarios, support for multi-tenancy as well as schemes to abstract users from the impact of multi-tenancy would be critical. However, explorations in this direction by companies such as Facebook [26], Microsoft [22, 38] and academic institutions neglect multi-tenant execution scenarios [47, 49]. However, the most relevant prior studies that have looked into multi-stage applications from the academic standpoint are as follows:

TimeTrader. [47] addresses the problem of meeting application specific latency targets for multi request execution in Online Data Intensive applications (OLDIs). Towards meeting that objective, they employ a mechanism that tries to reorder requests that contain varying slack using merely an **Earliest Deadline First** scheduling methodology. However, this technique assumes that the applications contain a single processing stage and fails to acknowledge the intrinsic latency variance across multiple stages. Hence, it deprioritizes requests assuming to contain relaxed latency constraints, however, would be subjected to a bulk of compute at its later stages. This leads to diminished effectiveness in mitigating response latency for multi-stage applications, as we quantitatively show in Section 6.

PowerChief. [49] seeks to identify the bottleneck stages present in multi-stage voice and image based intelligent personal assistant applications towards employing dynamic voltage frequency scaling to boost partial execution stages. However, PowerChief does not strive to guarantee SLAs at a request level. Furthermore, the proposed solution is not generalized for a microservice execution framework which handles requests from multiple tenants and focuses on a particular class of applications.

8 Conclusion

Microservice execution framework is rapidly transforming the operation of datacenters. It offers significantly more transparency into the underlying application execution than monolithic applications. Such visibility is a key enabler towards co-locating multiple latency critical applications on the same systems and still meeting SLAs. In the face of such visibility and changing opportunities, there is a clear need to rethink runtime systems and frameworks.

Towards this end, we present GrandSLAm, a runtime system that exploits this visibility along with identifying slack in individual queries of different applications. GrandSLAm enables multiple tenants to meet their SLAs, while achieving high throughput and utilization, with no performance overhead or programmer support. Therefore, we conclude that GrandSLAm can be an efficient substrate for current and future datacenter environments housing microservice execution frameworks.

Acknowledgment

We thank our anonymous reviewers for their constructive feedback and suggestions. This work was sponsored by the National Science Foundation (NSF) under NSF CAREER SHF-1553485. Jeongseob Ahn was supported by the National Research Foundation of Korea grant (NRF-2017R1C1B5075437) funded by MSIP, Korea.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*.

[2] Paula Aguilera, Katherine Morrow, and Nam Sung Kim. 2014. Fair share: Allocation of GPU resources for both performance and fairness. In *Proceedings of the IEEE 32nd International Conference on Computer Design (ICCD 14)*.

[3] Amazon. 2019. What is AWS Lambda? https://docs.aws.amazon.com/lambda/latest/dg/welcome.html. (2019).

[4] Amazon. 2019. What is AWS Step Functions? http://docs.aws.amazon.com/step-functions/latest/dg/welcome.html. (2019).

[5] Microsoft Azure. 2019. Azure Functions Serverless Architecture. https://azure.microsoft.com/en-us/services/functions/. (2019).

[6] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. 2013. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis lectures on computer architecture* 8, 3 (2013), 1–154.

[7] Quan Chen, Hailong Yang, Minyi Guo, Ram Srivatsa Kannan, Jason Mars, and Lingjia Tang. 2017. Prophet: Precise QoS Prediction on Non-Preemptive Accelerators to Improve Utilization in Warehouse-Scale Computers. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 17)*.

[8] Quan Chen, Hailong Yang, Jason Mars, and Lingjia Tang. 2016. Baymax: QoS Awareness and Increased Utilization for Non-Preemptive Accelerators in Warehouse Scale Computers. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 16)*.

[9] Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel P. Kuksa. 2011. Natural Language Processing (almost) from Scratch. *CoRR* abs/1103.0398 (2011). http://arxiv.org/abs/1103.0398

[10] Christina Delimitrou and Christos Kozyrakis. 2013. Paragon: QoS-aware Scheduling for Heterogeneous Datacenters. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 13)*.

[11] Tarek Elgamal, Atul Sandur, Klara Nahrstedt, and Gul Agha. 2018. Costless: Optimizing Cost of Serverless Computing through Function Fusion and Placement. *CoRR* abs/1811.09721 (2018). arXiv:1811.09721 http://arxiv.org/abs/1811.09721

[12] Sameh Elnikety, Erich Nahum, John Tracey, and Willy Zwaenepoel. 2004. A Method for Transparent Admission Control and Request Scheduling in e-Commerce Web Sites. In *Proceedings of the 13th International Conference on World Wide Web (WWW 04)*.

[13] A. Gheith, R. Rajamony, P. Bohrer, K. Agarwal, M. Kistler, B. L. White Eagle, C. A. Hambridge, J. B. Carter, and T. Kaplinger. 2016. IBM Bluemix Mobile Cloud Services. *IBM Journal of Research and Development* 60, 2-3 (March 2016), 7:1–7:12.

[14] Google. 2019. Serverless Environment to Build and Connect Cloud Services. https://cloud.google.com/functions/. (2019).

[15] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. 2017. Accurate, large minibatch SGD: training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677* (2017).

[16] Johann Hauswald, Yiping Kang, Michael A. Laurenzano, Quan Chen, Cheng Li, Ronald Dreslinski, Trevor Mudge, Jason Mars, and Lingjia Tang. 2015. Djinn and Tonic: DNN as a Service and Its Implications for Future Warehouse Scale Computers. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA 15)*.

[17] Johann Hauswald, Michael A. Laurenzano, Yunqi Zhang, Cheng Li, Austin Rovinski, Arjun Khurana, Ron Dreslinski, Trevor Mudge, Vinicius Petrucci, Lingjia Tang, and Jason Mars. 2015. Sirius: An Open End-to-End Voice and Vision Personal Assistant and Its Implications for Future Warehouse Scale Computers. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 15)*.

[18] Yuxiong He, Sameh Elnikety, James Larus, and Chenyu Yan. 2012. Zeta: Scheduling Interactive Services with Partial Execution. In *Proceedings of the Third ACM Symposium on Cloud Computing (SoCC 12)*.

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

[20] IBM. 2019. IBM Cloud Functions. https://www.ibm.com/cloud/functions. (2019).

[21] Muhammad Hussain Iqbal and Tariq Rahim Soomro. 2015. Big data analysis: Apache storm perspective. *International journal of computer trends and technology* 19, 1 (2015), 9–14.

[22] Virajith Jalaparti, Peter Bodik, Srikanth Kandula, Ishai Menache, Mikhail Rybalkin, and Chenyu Yan. 2013. Speeding Up Distributed Request-response Workflows. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*.

[23] Adwait Jog, Evgeny Bolotin, Zvika Guz, Mike Parker, Stephen W. Keckler, Mahmut T. Kandemir, and Chita R. Das. 2014. Application-aware Memory System for Fair and Efficient Execution of Concurrent GPGPU Applications. In *Proceedings of Workshop on General Purpose Processing Using GPUs (GPGPU 14)*.

[24] Evangelia Kalyvianaki, Marco Fiscato, Theodoros Salonidis, and Peter Pietzuch. 2016. THEMIS: Fairness in Federated Stream Processing Under Overload. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD 16)*.

[25] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. 2015. Profiling a Warehouse-scale Computer. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA 15)*.

[26] S. Kanev, K. Hazelwood, G. Y. Wei, and D. Brooks. 2014. Tradeoffs between power management and tail latency in warehouse-scale applications. In *IEEE International Symposium on Workload Characterization (IISWC 14)*.

[27] R. S. Kannan, A. Jain, M. A. Laurenzano, L. Tang, and J. Mars. 2018. Proctor: Detecting and Investigating Interference in Shared Datacenters. In *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS 18)*.

[28] O. Kayiran, N. C. Nachiappan, A. Jog, R. Ausavarungnirun, M. T. Kandemir, G. H. Loh, O. Mutlu, and C. R. Das. 2014. Managing GPU Concurrency in Heterogeneous Architectures. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 14)*.

[29] Kris Kobylinski. 2015. Agile Software Development for Bluemix with IBM DevOps Services. In *Proceedings of the 25th Annual International Conference on Computer Science and Software Engineering (CASCON 15)*.

[30] Jason Mars and Lingjia Tang. 2013. Whare-map: Heterogeneity in "Homogeneous" Warehouse-scale Computers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA 13)*.

[31] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. 2011. Bubble-Up: Increasing Utilization in Modern Warehouse Scale Computers via Sensible Co-locations. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 11)*.

[32] Sean Marston, Zhi Li, Subhajyoti Bandyopadhyay, Juheng Zhang, and Anand Ghalsasi. 2011. Cloud Computing - The Business Perspective. *Decis. Support Syst.* 51, 1 (April 2011), 14.

[33] David Meisner and Thomas F. Wenisch. 2012. DreamWeaver: Architectural Support for Deep Sleep. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 12)*.

[34] V. Nagarajan, R. Hariharan, V. Srinivasan, R. S. Kannan, P. Thinakaran, V. Sankaran, B. Vasudevan, R. Mukundrajan, N. C. Nachiappan, A. Sridharan, K. P. Saravanan, V. Adhinarayanan, and V. V. Sankaranarayanan. 2012. SCOC IP Cores for Custom Built Supercomputing Nodes. In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI 12)*.

[35] V. Nagarajan, K. Lakshminarasimhan, A. Sridhar, P. Thinakaran, R. Hariharan, V. Srinivasan, R. S. Kannan, and A. Sridharan. 2013. Performance and energy efficient cache system design: Simultaneous execution of multiple applications on heterogeneous cores. In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI 13)*.

[36] V. Nagarajan, V. Srinivasan, R. Kannan, P. Thinakaran, R. Hariharan, B. Vasudevan, N. C. Nachiappan, K. P. Saravanan, A. Sridharan, V. Sankaran, V. Adhinarayanan, V. S. Vignesh, and R. Mukundrajan. 2012. Compilation Accelerator on Silicon. In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI 12)*.

[37] Daniel Povey, Arnab Ghoshal, Gilles Boulianne, Lukas Burget, Ondrej Glembek, Nagendra Goel, Mirko Hannemann, Petr Motlicek, Yanmin Qian, Petr Schwarz, et al. 2011. The Kaldi speech recognition toolkit. In *IEEE 2011 workshop on automatic speech recognition and understanding*.

[38] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. 2014. A Reconfigurable Fabric for Accelerating Large-scale Datacenter Services. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA 14)*.

[39] Amazon Web Services. 2017. The Image Recognition and Processing Backend reference architecture demonstrates how to use AWS Step Functions to orchestrate a serverless processing workflow using AWS Lambda, Amazon S3, Amazon DynamoDB and Amazon Rekognition. https://github.com/aws-samples/lambda-refarch-imagerecognition. (2017).

[40] Samuel L Smith, Pieter-Jan Kindermans, and Quoc V Le. 2017. Don't Decay the Learning Rate, Increase the Batch Size. *arXiv preprint arXiv:1711.00489* (2017).

[41] Lavanya Subramanian, Vivek Seshadri, Arnab Ghosh, Samira Khan, and Onur Mutlu. 2015. The Application Slowdown Model: Quantifying and Controlling the Impact of Inter-application Interference at Shared Caches and Main Memory. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO 15)*.

[42] Lalith Suresh, Peter Bodik, Ishai Menache, Marco Canini, and Florin Ciucu. 2017. Distributed Resource Management Across Process Boundaries. In *Proceedings of the 2017 Symposium on Cloud Computing (SoCC 17)*.

[43] Shanjiang Tang, BingSheng He, Shuhao Zhang, and Zhaojie Niu. 2016. Elastic Multi-resource Fairness: Balancing Fairness and Efficiency in Coupled CPU-GPU Architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC 16)*.

[44] Prashanth Thinakaran, Jashwant Raj Gunasekaran, Bikash Sharma, Mahmut Taylan Kandemir, and Chita R Das. 2017. Phoenix: a constraint-aware scheduler for heterogeneous datacenters. In *IEEE 37th International Conference on Distributed Computing Systems (ICDCS 17)*.

[45] Prashanth Thinakaran, Jashwant Raj, Bikash Sharma, Mahmut T Kandemir, and Chita R Das. 2018. The Curious Case of Container Orchestration and Scheduling in GPU-based Datacenters. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC 18)*.

[46] T. Ueda, T. Nakaike, and M. Ohara. 2016. Workload characterization for microservices. In *IEEE International Symposium on Workload Characterization (IISWC 16)*.

[47] Balajee Vamanan, Hamza Bin Sohail, Jahangir Hasan, and T. N. Vijaykumar. 2015. TimeTrader: Exploiting Latency Tail to Save Datacenter Energy for Online Search. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO 15)*.

[48] Hailong Yang, Alex Breslow, Jason Mars, and Lingjia Tang. 2013. Bubble-flux: Precise Online QoS Management for Increased Utilization in Warehouse Scale Computers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA 13)*.

[49] Hailong Yang, Quan Chen, Moeiz Riaz, Zhongzhi Luan, Lingjia Tang, and Jason Mars. 2017. PowerChief: Intelligent Power Allocation for Multi-Stage Applications to Improve Responsiveness on Power Constrained CMP. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA 17)*.

[50] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache Spark: A Unified Engine for Big Data Processing. *Commun. ACM* 59, 11 (Oct. 2016), 10.

[51] Yilei Zhang, Zibin Zheng, and M.R. Lyu. 2011. Exploring Latent Features for Memory-Based QoS Prediction in Cloud Computing. In *IEEE Symposium on Reliable Distributed Systems (SRDS 11)*.