XX: A system

Anonymous Author(s)

Abstract

XXXXXX

1 Introduction

A world where cloud compute is run in the format of serverless jobs is attractive to developers and providers: developers pay only for what they use, while having access to many resources when needed; and cloud providers have control over scheduling and can use that to drive up utilization, rather than needing to hold idle resources available for clients who reserved them.

However, there remain realities that make serverless today fundamentally infeasible for workloads that in theory are a good fit — for instance web servers, which are often bursty and have inconsistent load, but are rarely run completely in serverless offerings [21], such as AWS lambda. One of the things that makes running a web server on lambda infeasible is lambda invocations' variable end to end latencies: in a small benchmark (broken down in Section 2) we found that total execution time latencies for a simple hello world function that sleeps for 20 ms ranged from 20 to 400ms.

An obvious and well-known problem that causes these high and variable latencies is that of cold starts. However, this paper takes the position that we are well underway to reaching low single digit ms cold start times, with current state-of-the-art research systems pushing into single digit territory [16, 19]. Which begs the question: if cold start is fast enought that more latency sensitive applications, like web servers, can have a cold start on the critical path, are we then done? Will serverless then be, at least from an infrastructure perspective, ready to support these sorts of workloads?

This paper argues that no, there still remains a fundamental obstacle to running such a latency sensitive workload on serverless: queueing and delay within the system. We cannot expect that the load will always fit in the resources providers have, and so some work has to be queued or otherwise degraded. In the world of long running servers, developers avoid degradation of access to resources by giving latency critical services reservations; but reserving resources in serverless doesn't work.

What developers care about in the end is that the jobs that are important run quickly. The problem we address is that, in a world where cold starts are fast and latency sensitive work is able running alongside map reduce and image processing jobs, important jobs might end up behind unimportant ones, waiting to be placed on machines and to get access to resources once placed. To address this problem, we propose XX, a scheduling system that has *price classes* as the central metric associated with each job. Priority in the system is directly paid for through price classes, and all of the resource allocation decisions in XX are made on the basis of price class.

XX has multiple goals it needs to achieve and challenges it needs to address.

One key goal is that XX needs to be able to support a multi tenant environment. This is enabled by the price classes being a universal metric. Rather than dealing in a relative ordering of developers' functions by importance, which would be difficult to compare across developers, the connection to money allows the class to have meaning in an absolute as well as a relative way. It also incentivizes usage of the lower classes for jobs that are less important, since they can be executed much more cheaply.

Another important goal is that of placing jobs quickly enough. For example, a job that takes 20ms to run cannot spend 50ms in scheduler queues and waiting for an execution environment before even starting to run. Even when they're short, cold starts will take up a significant portion of the time a job can handle waiting to run. So the challenge is knowing where the free and idle resources are, or finding out quickly, in a setting where both the number of new jobs coming in and the amount of resources are extremely large.

Finally, a key challenge in designing XX is that of managing memory. For compute, cores can always be timshared or processes preempted, but the buck stops once a machine is out of memory. This challenge is made more difficult by the fact that XX does not require developers to give memory usage limits for their jobs. This serves the purpose of extending the serverless on-demand structure to include memory, and moves away from the usual bin packing with overprovisioning problem. XX is thus faced with the challenging proposition of blindly placing jobs not knowing how much memory they will use, but still needing memory utilization to be high.

2 Motivation

We now explore a little more the benefits that serverless has to offer, the current state of the world, and how our approach fits in with both of these.

2.1 Benefits of serverless

The main attraction of serverless for developers is, in an idealized world, the characteristic of only paying for what you use while having a whole datacenter available to you. This is especially attractive to developers of applications where the

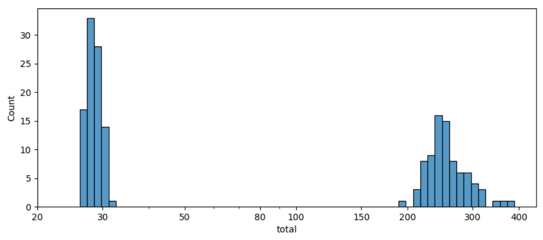

Figure 1: distribution of the total duration of the function. Note that the x axis is log scale

amount of resources that they need varies significantly over time, or is generally small and very spread out, so that buying their own machines or renting a fixed amount doesn't suit their actual needs well.

A central example to this paper is that of a web server. Its traffic patterns make it a great candidate for running entirely as serverless jobs: it is event-based, its load is bursty and unpredictable, and a request's resource requirements can vary greatly depending on which user invoked it.

Some back of the envelope math shows that for web servers with small load, lambda functions as they stand today are cheaper: for a low-traffic website, with approx 50K requests per day, a memory footprint of < 128 MB, and 200ms of execution, running that on AWS lambda adds up to \$1.58 per month. On the other hand, the cheapest EC2 instance costs just over \$3 per month. Of course, as the number of requests goes up, the price for lambdas scales linearly, whereas running an EC2 instance on full load becomes comparably cheap. There are pretty extensive simulations that others have done that show the tradeoff points for different types of workloads [7, 18].

Serverless also may outperform reservation systems for workloads that are very bursty: starting a new lambda execution environment is much faster than starting a new container or EC2 instance, which can take multiple minutes [1].

2.2 State of the world

However, only few web servers actually run entirely on serverless offerings today. There are many reasons that developers choose not to use serverless, despite in theory having workloads that are well-suited for the serverless environment [10, 20]. Popular complaints include provider lock in, lack of insight for debugging and telemetry, and variable runtimes.

We focus on what is arguably the most fundamental of these complaints; which is the variable runtimes. We run a small experiment with a hello world style lambda function that simply sleeps for 20ms and then returns. We use AWS Xray [2] to measure its latency, with incovations spaced randomly between 0 and 10 minutes. The results are in Figure 1.

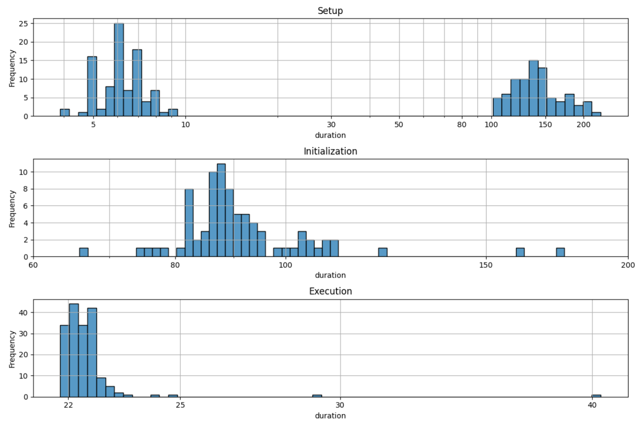

Figure 2: Breakdown of duration variation. Note that here too the x axes are all log scale

The times on the left side of the graph are clearly the execution times from warm start, which remain somewhat stable. The reason for this is likely that AWS is able to simply route the new request to the machine with the existing container on it. The right grouping in the graph is then the invocations that hit cold starts, whose overall latencies vary between 200 and 400ms, meaning that when a request encounters cold start, the variance in observed latency goes up a lot.

In order to better understand where that variability comes from, we look into a breakdown of the latencies. We are able to break down the total duration into three components: *setup*, which includes placing the function and creating the container; *initialization*, which is any code that runs before the function does, for example initializing the runtime, and any extensions the function uses; and *execution*, actually executing the functions code. Xray only gives us the latter two values explicitly, we calculate the setup time by looking at the total duration and subtracting the initalization and execution times.

We can see the resulting distributions in Figure 2. The most stable component of the duration is unremarkably the execution time, although with some outliers. The initialization also has a fair amount of variance, although there are no discernable groupings. The strongest variability, unsurprisingly, comes from the setup portion. Here we can see the two groupings clearly: one on the right that includes starting up the container, and one on the left that doesn't. The range is larger on the right (~150ms) than it is on the left (~8ms). Where exactly the latency here comes from is impossible to know without futher insight into the system, but clear is that there is already a small variation in warm start setting, where AWS doesn't even need to place a new container. And clear is also that sometimes AWS is able to get the cold start container up and running in just over 100ms, and sometimes it takes as much as ~250ms.

2.3 Approach

Our approach addresses the variability of runtimes, which is undesirable for latency sensitive jobs, by using price classes as a metric to tell the scheduler what to prioritize and what not. We will show that this allows us to stabilize the runtimes for the high class jobs.

In fact, we use classes to supplant completely the current interface, which requires developers choose an amount of memory per function, which is then tied to a cpu power (a potentially fractional amount of vCPUs). Price classes are a metric that has a number of benefits over resource reservations as an interface: developers are more likely to have a good sense of it ahead of time, it is less likely to be different across different invocations, it still gives the scheduler the information it needs to decide what to schedule when, and finally it more directly aligns the interests of the developer with those of the provider by communicating on the level of what the provider and developer actually care about: money, and latency (as achieved by classes in the system).

However, having price classes also means that there are no clear guarantees about what developers are getting when they put a price on a function they want to run. In order to mitigate that somewhat and not go into bidding wars, we propose exposing a fixed set of price classes. This is similar to how AWS has different EC2 instance types, that are directly mapped to prices. Rather than being a guarantee, the price class is instead a metric to express priority to XX, which it can then use to enforce a favoring of high class jobs.

3 Design

3.1 Interface

Developers using XX write function handlers and define triggers just like they would for any existing serverless offering. In addition, they asign each function to a price class, this is done at function creation. For instance, a simple web server might consist of a home page view that is assigned a higher price class and costs $2\mu¢$ per cpu second, a user profile page view which is assigned a middle-high class and cost $1.5\mu¢$ per cpu second, and finally an image processing job that can be set to a low class which costs only $0.5\mu¢$ per cpu second.

Classes are inherited across call chains: if a high class job calls a low class job, that invocation with run with high class. This is important in order to avoid priority inversion.

Developers pay for memory separately, and by use; the price for memory is the same across all classes.

To avoid unexpected costs in the case of for example a DOS attack or a bug, developers also express a monthly budget that they are willing to pay. XX uses this budget as a guideline and throttles invocations or decreases quality of service in the case that usage is not within reason given the expected budget, though it does not guarantee that the budget will not be exceeded by small amounts.

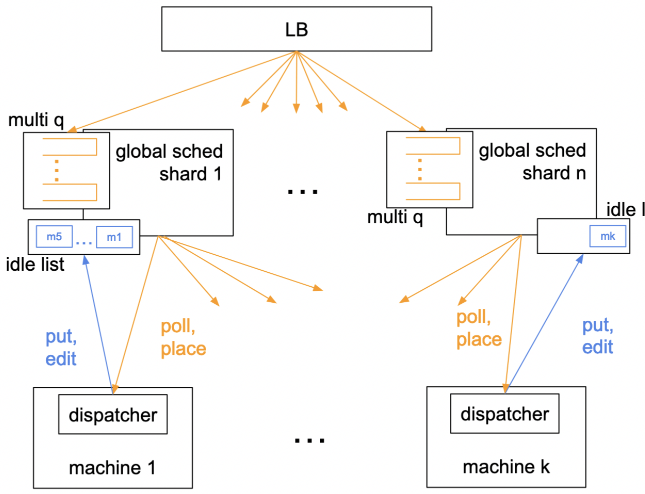

Figure 3: global scheduler shards queue and place jobs (in orange), on each machine a dispatcher thread keeps track of memory utilization and if it's low writes itself to an idle list (in blue)

3.2 XX Design

XX has as its goal to enforce the classes attached to jobs, which means it needs to prefer higher class jobs over lower ones, and preempt the latter when necessary.

As shown in Figure 3, XX sits behind a load balancer, and consists of: a *distributed global scheduler*, which places new job invocations and has attached an *idle list*, a *dispatcher*, which runs on each machine and communicates with the global scheduler shards, and a *machine scheduler*, which enforces classes on the machines.

Machine Scheduler. The machine scheduler is a preemptive priority scheduler: it preempts lower class jobs to run higher class ones. Being unfair and starving low class jobs is desirable in XX, since image processing jobs should not interrupt a page view request processing, but vice versa is expected. Within classes the machine scheduler is first come first served. This matches Linux' 'sched fifo' scheduling [3].

Idle list. Each global scheduler shard has an idle list, which holds machines that have a significant amount of memory available. In the shards idle list each machine's entry is associated with the amount of free memory as well as the current amount of jobs on the machine. The idle list exists because datacenters are large: polling a small number of machines has been shown to be very powerful, but cannot find something that is a very rare occurrence [14]. Having an idle list allows the machines that have actually idle resources, which are expected to be rare in a high-utilization setting, to make themselves visible to the global scheduler. The idle list also allows the global scheduler to place high class processes quickly, without incurring the latency overheads of doing the polling to find available resources. This is inspired by join idle queue [14], but centers memory rather than queue length.

Dispatcher. The dispatcher is in charge of adding itself to a shard's idle list when memory utilization is low. The dispatcher chooses which list to add itself to using power-of-k-choices: it looks at k shards' idle lists and chooses the one with the least other machines in it. If the machine is already on an idle list on shard i, but the amount of available memory has changed significantly (either by jobs finishing and memory being freed or by memory utilization increasing because of new jobs or memory antagonists), the dispatcher will update shard i's idle list. These interactions from the dispatcher to free lists are represented by the blue arrows in Figure 3.

The dispatcher is also in charge of managing the machine's memory. When memory pressure occurs, the dispatcher uses *class-based swapping* to move low class processes off the machine's memory. In this scenario, having priority scheduling creates the opportunity that enables this to be realistic: because the dispatcher knows that the lowest class jobs will not be run until the high class jobs have all finished, it can swap its memory out knowing it will not be needed soon. Avoiding a churn of jobs with swapped memory that are being swapped in and out as they are scheduled in and out, requires that the memory of the machine is large against the amount of memory that could be used by as many jobs as the machine has cores. This assumption ensures that memory pressure high enough to require swapping will only occur when there are many more jobs than there are cores, and thus that the swapped job will not be running soon. The dispatcher swaps the low class job back in when the memory pressure is gone and the job will be run.

Global Scheduler Shards. Global scheduler shards store the jobs waiting to be placed in a multi queue, with one queue per price class. Shards choose what job to place next by looking at each job at the head of each queue, and comparing the ratio of class to amount of time spent in the queue. This ensures that high class jobs don't have to wait as long as lower class jobs to be chosen next, but low class jobs will get placed if they have waited for a while.[hmng: why are we being fair-ish here when we're so unfair everywhere else? maybe just remove?]

When placing the chosen job, the shard will first look in its idle list. If the list is not empty, it will choose the machine with the smallest queue length.

If there are no machines in the idle list, the shard switches over to power-of-k-choices: it polls k machines, getting the amount of jobs running from each. The shard then places the new job on the machine with the smallest number of currently running jobs.

4 Preliminary Results

In order to understand the case for XX, we ask the following questions:

(1) How does job latency in XX compare to schedulers without priorities on one hand, and theoretically optimal schedulers with perfect information on the other?
(2) Does XX's plan for managing memory work?

To explore these questions, we built a simulator in go[4], which simulates different scheduling approaches.

4.1 Experimental Setup

In each version of the simulator, jobs arrive in an open loop at a constant rate. The simulator attaches three main characteristics to each job it generates: runtime, price class, and memory usage. *Job runtime* is chosen by sampling from randomly generated long tailed (in this case pareto) distribution: the relative length of the tail (α value) remains constant, and the minimum value (x_m) is chosen from a normal distribution. This reflects the fact that different functions have different expected runtimes (chosen from a normal distribution), and that actual job runtimes follow long tailed distributions (so each pareto distribution that we sample represents the expected runtime distribution of a given function). *Job class* is chosen randomly, but weighted: the simulator uses a vaguely bimodal weighting across priorties. The simulator has n different price class values, each assigned to a fictitious price. Because functions are randomly assigned a class, runtime and class are not correlated. *Job memory usage* is chosen randomly between 100MB and 10GB. Over their lifetime, jobs increase their memory usage from an initial amount (always 100MB) to their total usage.

When comparing two different simulated schedulers, they each are given an identical workload and then each simulate running that workload.

The simulator makes some simplifying assumptions:

(1) functions are compute bound, and do not block for i/o
(2) communication latencies are not simulated
(3) the amount of time it takes to swap memory is not simulated

We simulate running 100 machines with 8 cores each, 4 scheduler shards, and a k-choices value of 3 when applicable.

4.2 How do job latencies compare?

In the end developers care about job latency, so it is important to understand how well price classes do at reflecting and enforcing SLAs. On one hand, is relevant to understand if we need classes at all: is there a scheduler that can, without having any access to information about which jobs are important, still ensure that jobs perform well? On the other

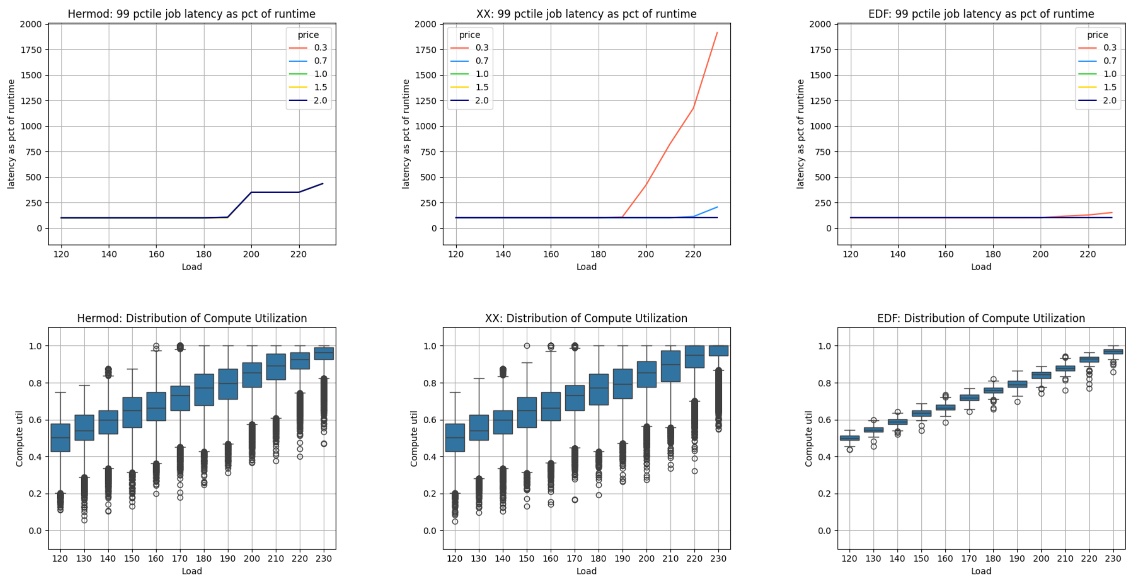

Figure 4: tail latency distribution and compute utilization across different loads, for Hermod, XX, and EDF respectively

hand, it is helpful to compare XX to an ideal scheduler, in order to contextualize XX's performance.

To explore one side of this question, we look at the performance of an existing scheduler that does not take any form of priority into account. We simulate Hermod[13], a state-of-the-art research scheduler built specifically for serverless. Hermod's design is the result of a from-first-principles analysis of different scheduling paradigms. In accordance with the paper's findings, we simulate least-loaded load balancing over machines found using power-of-k-choices, combined with early binding and Processor Sharing machine-level scheduling. Hermod does not use priorities in its design, and as such the simulator ignores jobs' class when simulating Hermod's design.

On the other side, we want to simulate an ideal scheduler. Ideal here is with respect to meeting jobs' SLAs, which requires defining the desired SLA. In order to do this, we assign each job invocation a deadline, and allow the deadline to be a function of the job's true runtime. We define the deadline as a function of the runtime as well as the price class, as follows: deadline = runtime * maxPrice/price. This ensures that the highest classes jobs' deadlines are simply their runtimes, and deadlines get more and more slack with lower classes. We then simulate an Earliest Deadline First (EDF) scheduler over these deadlines, which is queuing theoretically proven to be optimal in exactly the way we wanted: if it is possible to create a schedule where all jobs meet their deadline, EDF will find it[5]. We run EDF in a centralized setting (it schedules one machine with 800 cores, not 100 machines with 8 each), in order to avoid complicated placement decisions based on deadline schedules.

We compare the latencies observed in both of these settings with those that running XX produces. Because Hermod does not talk about dealing with memory pressure, and to avoid an unfair comaprison with XX's swapping, we set the memory to be absurdly high for all three settings in this experiment. We also turn off the use of the idle list in XX, so as to be en par with Hermod in placing load, and revert solely to k-choices.

A strong result for XX would show that its performance is between the two, and closer to the EDF side. Especially as load and utillization get high, we expect that the differences betweeen the three approaches will become evident. Figure 4 shows the results. We can see that XX and EDF are able to retain performance for the high class jobs even under higher load. EDF's superior performance on the far right side of the graph for the low class jobs is becuase EDF only runs those jobs with short runtimes; the others never finish and so don't show up on this graph. EDF's utilization is also much tighter, because of the centralised setting it runs in (so the spread of the utilization comes from being different at different points in time, not on different machines).

4.3 Does XX plan for memory management work?

To answer this question, we look at how XX distributes load, and whether the amount that XX needs to swap memory is realistic. We now run XX in a setting of limited memory (32GB of RAM per machine), and track the memory utilization of different machines, as well as how much and what

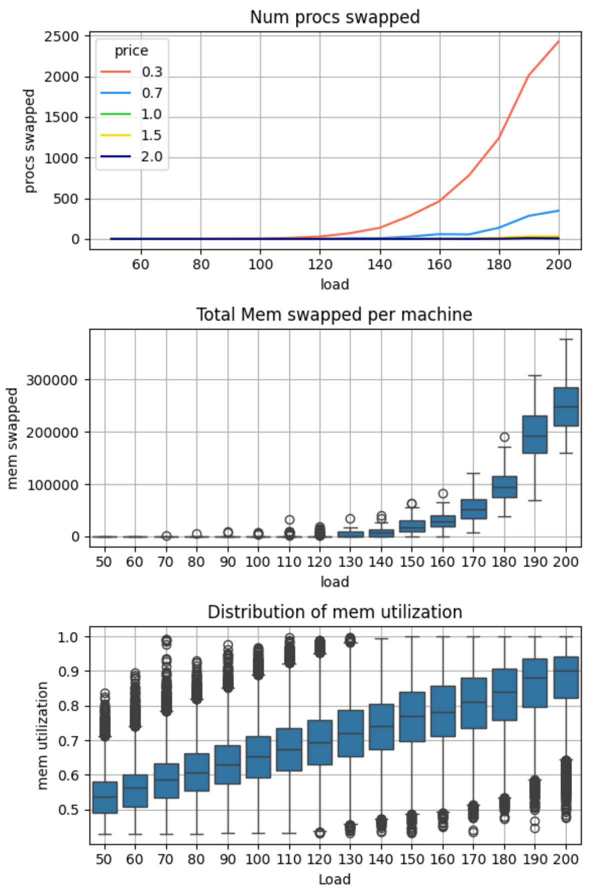

Figure 5: XX's swapping behavior. The amount of memory is in MB

they need to swap. A good result would show a tigh spread of memory utilization, that machines only start swapping once memory utilization is high, and that the amount of swapping being done is also equally spread across machines. Figure 5 shows the results. We can see we only swap lower class jobs' memory, and that the amount of memory swapped is faily evenly distributed between all the machines.

5 Discussion

There remain many interesting open questions in fully fleshing out this problem and design. We discuss some of the questions that we expect will guide our future research on this project; the first two are somewhat concrete technical questions and the last one is more of a philosophical musing about the chosen interface.

Is swapping the right thing to be doing? In our design we use swapping, which works because of the assumption we make about memory being large enough to for sure hold the memory of all the processes that could fit on the cores. This assumption is likely true for the active working sets of most jobs, but potentially not for their whole memory. If we lift the assumption, then we are left with a potentially very undesirable situation where less jobs than there are cores fit in the memory, and so they are constantly being swapped in and out as the idle core wants to run the job that was swapped out, and swaps out a job actually running somewhere else.

In order to avoid this situation, paging, or a combination of swapping and paging, might be better. Some of the challenges in answering this question are choosing what memory to page, and deciding how billing works for jobs who were forced to page some of their memory.

Can we bound the amount of memory we need to swap/page? Given some swapping and/or paging scheme, there is a fundamental question of whether we can bound the amount of memory that would need to be swapped/paged on any machine. This is especially tricky in combination with the fact that we do not have any sense of how much memory each job will use. One potential assumption that would help here is to bound the maximum amount of memory that any job can use; the bound could be very high, but doing so would allow us to use queue length to also bound the amount of memory that the machine will need to have (in both RAM and swapspace).

How do developers know they not are getting cheated? In our scheme based on price classes, there are no outwardly visible guarantees in the same way that there are for resources (where running a short benchmark will show exactly the amount of resources you have access to). There are however many other ways that we already trust cloud providers without much proof that we can verify ourselves, including that fundamentally providers could always simply deny service and stop running anything. Of course providers have market-based incentive to not provide lesser quality services for the same price, and certainly a coordinated effort would quickly be able to reveal if priority inversions were in fact going on.

6 Related Work

Many other projects have explored how to do better serverless scheduling.

Systems like Sparrow[15], Hermod[13], or Kairos[9] improve performance of scheduling in the distributed setting by trying out and using different scheduling policies. Unlike XX, they treat all jobs equally.

Other approaches generate information about jobs that are coming in to help placement decisions; for instance ALPS[11], which observes and learns the behaviors of existing functions and then makes scheduling decisions based on those; or Morpheus[12], which learns SLOs from historical runs, and then translates these to recurring reservations. XX instead gets the classes directly from the developers as part of its interface.

Other papers have taken the same approach as XX of getting information to help scheduling from the developers. Sequoia[17], for instance, creates a metric of QOS for serverless functions. Unlike XX however, Sequoia does not implement a new scheduler, but builds a layer in front of AWS lambda.

Another project[8] creates a language of Allocation Priority Policies (APP), which is a declarative language to express policies, then builds a scheduler around that. Unlike XX, the APP language is built around making load balancing decisions and ultimately defines a mapping of jobs to workers, rather than associating priorities with the jobs and having the scheduler do the scheduling automatically.

AWS lambda itself also goes this route, by offering two different ways for developers to influence lambda function scaling: provisioned and reserved concurrency[6]. Provisioned concurrency specifies a number of instances to keep warm, and reserved concurrency does the same but ensures that those warm instances are kept for a specific function. However, these knobs are more centered around mitigating the effects of cold start rather than actually affecting the way the jobs are scheduled.

References

[1] URL: https://docs.aws.amazon.com/autoscaling/ec2/userguide/ec2-auto-scaling-default-instance-warmup.html.

[2] URL: https://aws.amazon.com/xray/.

[3] URL: https://man7.org/linux/man-pages/man7/sched.7.html.

[4] URL: https://go.dev/.

[5] URL: https://en.wikipedia.org/wiki/Earliest_deadline_first_scheduling.

[6] URL: https://docs.aws.amazon.com/lambda/latest/dg/lambda-concurrency.html.

[7] Álvaro Alda Rodríguez et al. *Economics of 'Serverless'*. URL: https://www.bbva.com/en/innovation/economics-of-serverless/.

[8] Giuseppe De Palma, Saverio Giallorenzo, Jacopo Mauro, and Gianluigi Zavattaro. "Allocation Priority Policies for Serverless Function-Execution Scheduling Optimisation". In: *Service-Oriented Computing: 18th International Conference, ICSOC 2020, Dubai, United Arab Emirates, December 14–17, 2020, Proceedings*. Dubai, United Arab Emirates: Springer-Verlag, 2020, pages 416–430.

[9] Pamela Delgado, Diego Didona, Florin Dinu, and Willy Zwaenepoel. "Kairos: Preemptive Data Center Scheduling Without Runtime Estimates". In: *Proceedings of the ACM Symposium on Cloud Computing*. SoCC '18. Carlsbad, CA, USA: Association for Computing Machinery, 2018, pages 135–148.

[10] Jesse Duffield. *My notes from deciding against AWS Lambda*. URL: https://jesseduffield.com/Notes-On-Lambda/.

[11] Yuqi Fu, Ruizhe Shi, Haoliang Wang, Songqing Chen, and Yue Cheng. "ALPS: An Adaptive Learning, Priority OS Scheduler for Serverless Functions". In: *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. Santa Clara, CA: USENIX Association, July 2024, pages 19–36.

[12] Sangeetha Abdu Jyothi, Carlo Curino, Ishai Menache, Shravan Matthur Narayanamurthy, Alexey Tumanov, Jonathan Yaniv, Ruslan Mavlyutov, Inigo Goiri, Subru Krishnan, Janardhan Kulkarni, and Sriram Rao. "Morpheus: Towards Automated SLOs for Enterprise Clusters". In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA: USENIX Association, Nov. 2016, pages 117–134.

[13] Kostis Kaffes, Neeraja J. Yadwadkar, and Christos Kozyrakis. "Hermod: principled and practical scheduling for serverless functions". In: *Proceedings of the 13th Symposium on Cloud Computing*. SoCC '22. San Francisco, California: Association for Computing Machinery, 2022, pages 289–305.

[14] Yi Lu, Qiaomin Xie, Gabriel Kliot, Alan Geller, James R. Larus, and Albert Greenberg. "Join-Idle-Queue: A novel load balancing algorithm for dynamically scalable web services". In: *Performance Evaluation* 68.11 (2011). Special Issue: Performance 2011, pages 1056–1071.

[15] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. "Sparrow: distributed, low latency scheduling". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP '13. Farminton, Pennsylvania: Association for Computing Machinery, 2013, pages 69–84.

[16] Ariel Szekely, Adam Belay, Robert Morris, and M. Frans Kaashoek. "Unifying serverless and microservice workloads with SigmaOS". In: *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*. SOSP '24. Austin, TX, USA: Association for Computing Machinery, 2024, pages 385–402.

[17] Ali Tariq, Austin Pahl, Sharat Nimmagadda, Eric Rozner, and Siddharth Lanka. "Sequoia: enabling quality-of-service in serverless computing". In: *Proceedings of the 11th ACM Symposium on Cloud Computing*. SoCC '20. Virtual Event, USA: Association for Computing Machinery, 2020, pages 311–327.

[18] Andy Warzon. *AWS Lambda Pricing in Context - A Comparison to EC2*. URL: https://www.trek10.com/blog/lambda-cost.

[19] Xingda Wei, Fangming Lu, Tianxia Wang, Jinyu Gu, Yuhan Yang, Rong Chen, and Haibo Chen. "No Provisioned Concurrency: Fast RDMA-codesigned Remote Fork for Serverless Computing". In: *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. Boston, MA: USENIX Association, July 2023, pages 497–517.

[20] *Why or why not use AWS Lambda instead of a web framework for your REST APIs?* URL: https://www.reddit.com/r/Python/comments/1092py3/why_or_why_not_use_aws_lambda_instead_of_a_web/.

[21] *Without saying "it's scalable", please convince me that a serverless architecture is worth it.* URL: https://www.reddit.com/r/aws/comments/yxyyk3/without_saying_its_scalable_please_convince_me/.