XX: A system

Anonymous Author(s)

1 Introduction

A world where cloud compute is run in the format of serverless functions 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.

There remain roadblocks, however, that make serverless today infeasible for workloads that are a good fit. For instance web server applications, which often have inconsistent and bursty load, but are rarely run completely in serverless offerings [34], such as AWS lambda. One of the challenges is lambda invocations' variable end to end latencies: in a small benchmark (described 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; whereas an acceptable latency for user-facing pages is be anything below 100ms [23]. This variability is a problem because it has been shown that small response time differences can matter a lot in interactive applications [12, 17].

A well-known cause of these variable latencies is cold starts. This paper takes the position that systems research is well underway to reaching low single digit ms cold start times, with current state-of-the-art research systems pushing into single digit territory [29, 32]. Which begs the question: if cold start is fast enought that more latency sensitive applications, like web applications, 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 challenge to running such a latency sensitive workload on serverless: queueing and delay within the system. Load will not always fit in the resources providers have, and so some work msut 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 servers is incompatible with the serverless approach.

What developers care about in the end is that the functions that are latency sensitive run quickly. This paper assumes a world where cold starts are fast and latency sensitive work runs alongside map reduce and image processing functions. The challenge this paper addresses is that latency sensitive functions might end up behind background ones, waiting to be placed on machines or to get access to resources. To address this challenge, this paper proposes XX, a scheduling system that associates with each function a *price class*, which is an amount that it costs to run that function per unit time. 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 goal is that XX needs to be able to support a multi tenant environment. Price classes achieve this: rather than dealing in a relative ordering of developers' functions by latency sensitivity, which would be difficult to compare across developers, the connection to money allows the price class to have meaning on an absolute scale. It also incentivizes usage of the lower price classes for functions that are less latency sensitive, since they can be run cheaply.

Another important goal is that of placing functions quickly enough. For example, a function that takes 20ms to run cannot spend 50ms in scheduler queues and waiting for an execution environment before even starting to run. Knowing where the free and idle resources are, or finding out quickly, is challenging in a setting where both the number of new functions invocations and the amount of resources are large.

Finally, a key challenge in designing XX is that of managing memory. For compute resources, cores can be timshared or processes preempted, but the buck stops once a machine is out of memory. Current systems address this challenge by requiring developers to express a maximal amount of memory they will use, and charging based on that. However, memory usage is at best difficult to know in advance and at worst has a large variance so is impossible to say in advance. And more importantly, is not correlated with what developers actually care about, which is function latency. Instead, XX charges developers based on the amount of memory actually used, and requires no bound to be set. XX is thus faced with the challenging proposition of blindly placing functions not knowing how much memory they will use, but still needing memory utilization to be high.

2 Motivation

This paper is motivated by the benefits of serverless as an approach to utility computing, and finds latency variability to be a key challege in making true serverless a reality.

2.1 Benefits of serverless

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

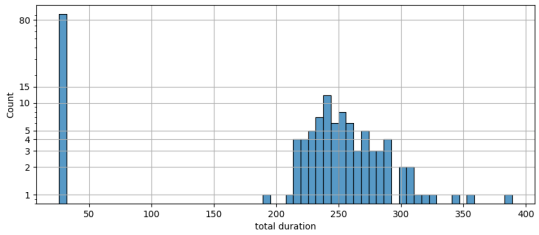

Figure 1: distribution of end to end duration times. The y axis is log scale

proposition is especially attractive to developers of applications where the amount of resources that they need varies significantly over time, or is generally small and very spread out. With such workloads, buying their own machines or renting a fixed amount of server space is bad for the developer because it is expensive if provisioned for peak usage and has poor performance if not, and bad for providers because it leads to low utilization.

A central example to this paper is that of a web server. Its traffic patterns make it a great candidate for running as serverless functions: 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.

A back of the envelope calculation 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. Extensive simulations show a more nuanced picture of the tradeoff points for different workloads [11, 31].

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 Challenge of latency variability

However, only few web applications run entirely on serverless offerings today. There are many reasons that developers choose not to use serverless, despite their workloads being well-suited for the serverless environment [15, 33]. Popular complaints include provider lock in, lack of insight for debugging and telemetry, and variable runtimes.

XX focuses on the challenge of variable runtimes. In order to better understand the where the variation comes from, we run an experiment with a simple lambda function that 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. The spike on the left side of the graph is the execution times from invocations that used warm start. We can see the durations remain stable. The reason for this is that AWS is able to simply route the new request to the machine with the existing container on it. We verify that this is indeed what is happening by changing the function to include reading and then writing to an environment variable, and find that for invocations with warm start when we read the variable it was already set by a previous invocation.

The right grouping in the graph is those invocations that hit cold starts, whose overall latencies vary between ~200 and ~400ms.

In order to understand where the extra latency in the longer times comes from, we begin by ruling out some options such as container downloading, and then look to open source alternatives to come to the conclusion that it comes from queueing or delay (waiting for resources) in the system.

???

[hmng: What we are trying to show here is that the stable latencies from warm start wouldn't spill over into the cold start world even if cold start is fast? Or maybe a flipside of that which is that part of the cold start latency comes from queuing already now? Those feel like two different things — one is more the breakdown thing, the other is more the openwhisk experiment with everything warm start]

Indeed, it must be the case that any system that runs with high average utilization experiences moments where there is more load than the resources can handle. As we know from recent traces [35], although load may look stable at a time increment of hours or even minutes, going down to the second and millisecond level shows the load to have high variance. If providers want to have good average utilization, then in the moments of load spike the load will be more than the resources can handle.

The scheduler then has two different options: queue the excess load, or place it on machines and let them deal with being temporarily overloaded. Different schedulers have different approaches. In OpenWhisk [26], for instance, the load balancer will choose which machine to run the function on, and then place the invocation, adressed to that machine, into a Kafka queue that the machine subscribes to and can pull the invocation from when it is ready [3]. Knative [20] similarly queues the excess invocations, although it does so via the load balancer, which is also in charge of autoscaling [21]: if the existing pods are fully loaded (with a small, bounded-size queue in front of them), requests are queued separately while the autoscaler starts up more invocations. It is impossible to know what proprietary systems like AWS do, but since AWS guarantees an amount of memory as well as a fraction of vCPUs to each function, it is likely AWS also queues the invocations to ensure they aren't put in a position of having

to break those guarantees.[hmng: that might be a test we could run: do a tight while loop and look at cpu time and wall clock time to see how we are being scheduled]

Because none of these schedulers have information about the functions they are queueing, it is impossible for them to know which to prioritize. Without further tooling, it is possible and in fact likely that latency sensitive jobs might end up in the queue behind background functions. The way all of the above schedulers deal with this is by doing some sort of accounting of concurrency: concurrency can be reserved or provisioned for specific functions, and limited for others. This is necessary to ensure that a burst in background tasks doesn't starve the latency sensitive functions. However, what happens when the datacenter is out of resources and so the concurrency limit has not yet been reached but the resources are unavailable is not clear. Reserving and provisioning and limiting are also conceptually in tension with the goal of serverless, which is to be on-demand and flexible.

3 Approach & Design

This paper's approach addresses the variability of runtimes, which is undesirable for latency sensitive functions, by using price classes as a metric to tell XX what to prioritize and what not. We will show that having price classes allows XX to stabilize the runtimes for high price class functions.

XX uses price classes to supplant the current interface, which requires developers to choose an amount of memory per function, which is tied to a cpu fraction (e.g., 0.2 vCPUs). In XX, price classes are the only thing that developers need to give; XX bills memory separately and by use. The price for memory is the same across all price classes. Removing memory from the interface serves the purpose of extending the serverless on-demand structure to include memory.

Price classes are a metric that has a number of benefits over resource usage estimations. One is that developers are more likely to have a good sense of what price class a function should have ahead of time, because they know in what context the function will be used and how important it is that the function run quickly. Price classes also remain the same across different invocations, whereas the resources needed can be heavily skewed in a web application environment, where popularity distributions are often very uneven [19, 28]. And finally, price classes more directly align 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 price classes in the system).

However, having price classes also means that there are no absolute guarantees about what developers are receiving when they put a price on a function. In order to mitigate that and avoid the developer-side uncertainty of bidding wars, XX exposes a fixed set of price classes. This fixed price list 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 XX uses to enforce a favoring of high price class functions.

Having price classes also allows the provider to provision their datacenters in terms of the amount of hardware they buy: by looking at the historical overall amount of high price class load, they know a minimum of how much hardware they need to buy to be able to comfortably fit that load.

At the same time, price classes give XX the information it needs to decide what to schedule when. How exactly it does this is what we explore in XX's design.

3.1 Interface

Developers using XX write function handlers and define triggers just like they would for any existing serverless offering. In addition, each place where they trigger the function, they assign that invocation to a price class. For instance, a simple web application 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 price class and cost $1.5\mu¢$ per cpu second, and finally an image processing function that can be set to a low price class which costs only $0.5\mu¢$ per cpu second.

Price classes are inherited across call chains: if a high price class function calls a low price class function, that invocation with run with high price class. This inheritance is important in order to avoid priority inversion.

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.

3.2 XX Design

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

As shown in Figure 2, XX sits behind a load balancer, and consists of: a *distributed global scheduler*, which places new function 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 price classes on the machines.

Machine Scheduler. The machine scheduler is a preemptive priority scheduler: it preempts lower price class functions to run higher price class ones. Being unfair and starving low price class functions is desirable in XX, since image processing functions should not interrupt a page view

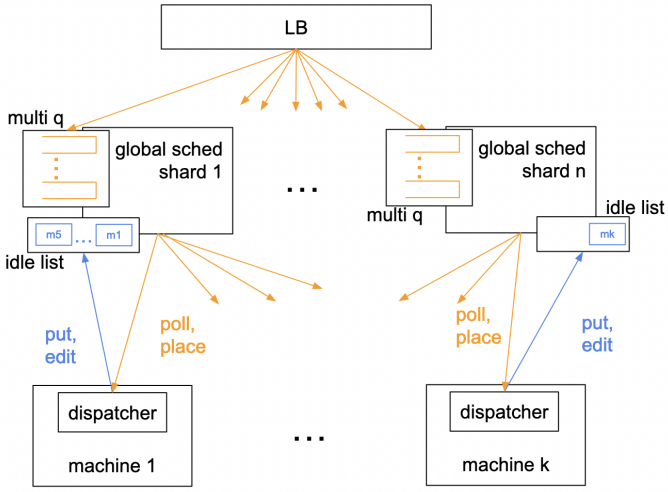

Figure 2: global scheduler shards queue and place functions (in orange), on each machine a dispatcher thread keeps track of memory utilization and if it's low writes itself to an idle list (in blue)

request processing, but vice versa is expected. Within price classes the machine scheduler is first come first served. This design matches Linux' 'SCHED FIFO' scheduling [4].

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 memory available as well as the current amount of functions 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 [24]. 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 price class functions quickly, without incurring the latency overheads of doing the polling to find available resources. This design is inspired by join idle queue [24], but defines idleness via memory availability rather than empty queues.

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 functions finishing and memory being freed or by memory utilization increasing because of new functions 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 2.

The dispatcher is also in charge of managing the machine's memory. When memory pressure occurs, the dispatcher uses *price class-based swapping* to move low price class functions off the machine's memory. Having priority scheduling creates an opportunity: because the dispatcher knows that the lowest price class functions will not be run until the high price class functions have all finished, it can swap its memory out knowing it will not be needed soon. The dispatcher swaps the low price class function back in when the memory pressure is gone and the function will be run.

Bounding the amount of swap space required without bounding the amount of memory that functions can use is not possible. The goal of the dispatcher is to swap when possible, and in the case that that is not enough it can resort to killing. Providers can estimate the amount of swap space required by looking at memory utilization, and since swap space is cheap [5] can provision it so that killing is very rare.

Global Scheduler Shards. Global scheduler shards store the functions waiting to be placed in a multi queue, with one queue per price class. Shards choose what function to place next by looking at each function at the head of each queue, and comparing the ratio of price class to amount of time spent in the queue. This procedure ensures that high price class functions don't have to wait as long as low price class functions to be chosen next, but low price class functions will get placed if they have waited for a while.

When placing the chosen function, 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 functions running from each. The shard then places the new function on the machine with the smallest number of currently running functions.

4 Preliminary Results

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

(1) How does function 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 build a simulator in go[6], which simulates different scheduling approaches. Using a simulator allows us to extend the experiments to include many more machines than would otherwise be available to us.

4.1 Experimental Setup

In each version of the simulator, functions arrive in an open loop at a constant rate. The simulator attaches three main

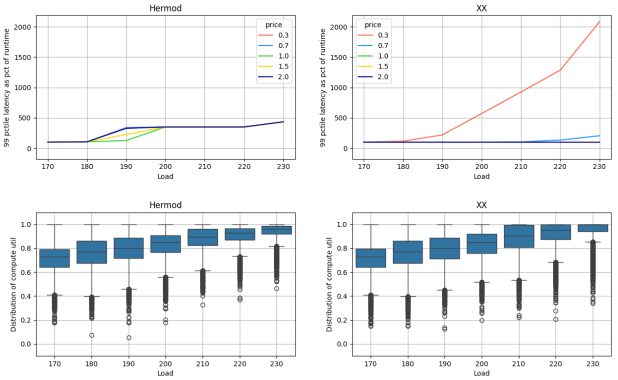

Figure 3: tail latency distribution and compute utilization across different loads, for Hermod and XX

characteristics to each function it generates: runtime, price class, and memory usage. *Function 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 process for sampling reflects the fact that different functions have different expected runtimes (chosen from a normal distribution), and that actual invocation runtimes follow long tailed distributions (so each pareto distribution that we sample represents the expected runtime distribution of a given function). *Function price class* is chosen randomly, but weighted: the simulator uses a bimodal weighting across priorties. The simulator has n different price class values, each assigned to a fictitious price. Because functions are randomly assigned a price class, runtime and price class are not correlated. *Function memory usage* is chosen randomly between 100MB and 10GB. Over their lifetime, functions 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 function latencies compare?

Developers care about function 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 price classes at all: is there a scheduler that can, without having any access to information about which functions are latency sensitive, still ensure that functions perform well? On the other hand, it is helpful to compare XX to an ideal scheduler, in order to contextualize XX's performance.

To explore the first side of this question, we look at the performance of an existing scheduler that does not take priority into account. We simulate Hermod[19], 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 functions' price class when simulating Hermod's design.

On the other side, we want to simulate an ideal scheduler. Ideal is with respect to meeting functions' SLAs, which requires defining the desired SLA. In order to do this, we assign each function invocation a deadline, which we define as a function of the true runtime as well as the price class: deadline = runtime * maxPrice/price. This definition ensures that the highest price classes functions' deadlines are simply their runtimes, and deadlines get more and more slack with lower price 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 functions meet their deadline, EDF will find it[7]. We run EDF in a centralized setting: it schedules one machine with 800 cores, not 100 machines with 8 each.

We compare the latencies observed in both of these settings with those that running XX produces. Because Hermod does not deal 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 on par with Hermod in placing load, and revert solely to k-choices.

A strong result for XX would show that it is able to maintain low latency for high price class functions. Especially under high load, we expect that the differences betweeen the three approaches will become evident. Figure 3 shows the results. We can see that XX and EDF are able to retain performance for the high price class functions even under higher load. EDF's superior performance on the far right side of the graph for the low price class functions is because EDF runs only functions 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).

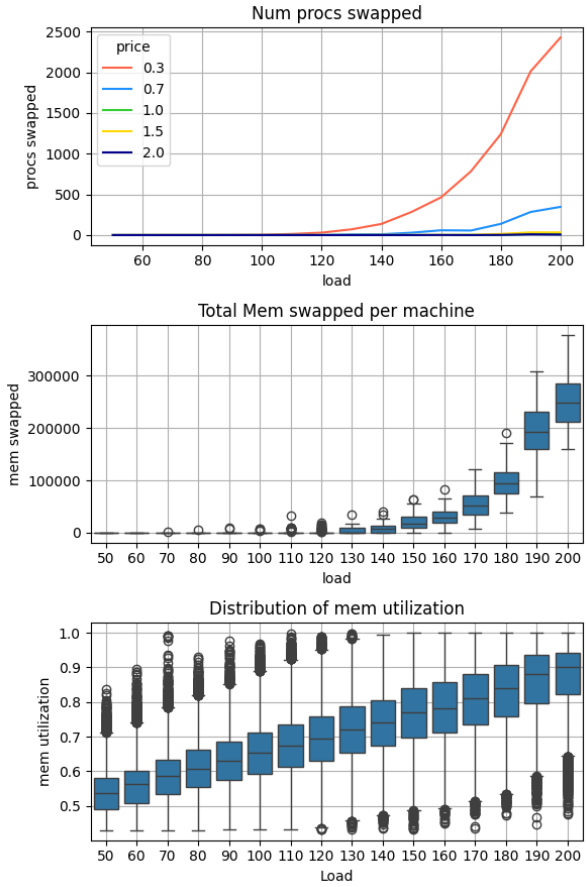

Figure 4: XX's swapping behavior. The amount of memory is in MB

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 machines need to swap. A good result would show: a small spread of memory utilization, that machines only start swapping once memory utilization is high, and that the amount of swapping being done is equally spread across machines. Figure 4 shows the results. We can see XX swaps only lower price class functions' memory, and that the amount of memory swapped is faily evenly distributed between all the machines. We can also conclude that with a 500GB SSD, a provider would comfortably be able to avoid killing while running the datacenter at an average memory utilization of ~90%, at the cost of ~ $30 per machine for swap space [5].

5 Related Work

Many other projects have explored how to do better scheduling for data centers.

Systems like Sparrow[27], Hermod[19], or Kairos[14] improve performance of scheduling in the distributed setting by trying out and using different scheduling policies. Unlike XX, they treat all functions equally.

Like XX, many projects tailor their approach to serverless. Some systems generate information about functions themselves to help placement decisions; for instance ALPS[16], which observes and learns the behaviors of existing functions and then makes scheduling decisions based on those; or Morpheus[18], which learns SLOs from historical runs, and then translates these to recurring reservations. XX instead gets the price 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[30], for instance, creates a metric of QOS for serverless functions. Unlike XX however, Sequoia does not implement a new scheduler, but is itself a serverless function that manages the invocation sequence of developer's function chains by interposing on the triggers and choosing what to invoke when. Therefore it also, unlike XX, does not support multi-tenancy.

Allocation Priority Policies (APP)[13] provides a declarative language to express policies, then builds a scheduler around that. The APP language is built around allowing developers to specify custom load balancing decisions, and the scheduler uses the developers' specification to define a mapping of function invocations to workers. XX, on the other hand, does not ask developers to set the load balancing policy, but rather has developers give XX the information it needs to do the load balancing itself.

AWS offers two different ways for developers to influence their functions' scaling: provisioned and reserved concurrency[8]. Provisioned concurrency specifies a number of instances to keep warm for a given function, and reserved concurrency ensures that a fixed amount of the possible concurrency reserved for it. This interface is bad for serverless workloads for the same reasons that reservation-based systems are: it requires developers to estimate their future needs and pay up front, and providers to keep those potentially idle resources available.

On the serverful side of scheduling, priorities are generally expressed via a latency critical/best effort binary, where latency critical processes have an attached amount of resource reservations, and best effort processes don't [22]. Serverless schedulers that sit on top of Kubernetes, such as OpenFaaS [25] or Knative [20], use autoscaling to keep up with load, and use the same interface as kubernetes for

developers to express resource requirements [9, 10]. This works well for long running servers with steady amounts of load, since predictable load will allow developers to make good approximations of the resources they will need. The serverless setting XX works within is different because both the number of invocations and the resource usage of each invocation is expected to vary.

6 Conclusion

Serverless was and is a great option for developers whose load varies and providers who don't want to keep resources idle for processes that reserved them. However, the reality of serverless today is that functions experience a variance in latencies that is not tolerable for latency sensitive workloads.

We propose a new scheduler, XX, that introduces *price classes*. Developers assign each function they want to run to a price class, which encodes a priority that XX then enforces at invocation, both in placing the function and on the machine level by running priority scheduling. We show that XX is able to enforce priorities and keep high price class functions latencies stable even under high load.

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://github.com/apache/openwhisk/blob/master/docs/about.md.

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

[5] URL: https://www.bestbuy.com/site/pny-cs900-500gb-internal-ssd-sata/6385542.p?skuId=6385542.

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

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

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

[9] URL: https://knative.dev/docs/serving/services/configure-requests-limits-services/.

[10] URL: https://docs.openfaas.com/reference/yaml/.

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

[12] Jake Brutlag. *Speed Matters*. URL: https://research.google/blog/speed-matters/.

[13] 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.

[14] 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.

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

[16] 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.

[17] Gigaspaces. *Amazon Found Every 100ms of Latency Cost them 1 Percent in Sales*. URL: https://www.gigaspaces.com/blog/amazon-found-every-100ms-of-latency-cost-them-1-in-sales.

[18] 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.

[19] 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.

[20] *Knative*. URL: https://knative.dev/.

[21] Stavros Kontopoulos. *Demystifying Activator on the data path*. URL: https://knative.dev/blog/articles/demystifying-activator-on-path/.

[22] Kubernetes. *Configure Quality of Service for Pods*. URL: https://kubernetes.io/docs/tasks/configure-pod-container/quality-service-pod/.

[23] Greg Linden. *Marissa Mayer at Web 2.0*. URL: http://glinden.blogspot.com/2006/11/marissa-mayer-at-web-20.html.

[24] 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.

[25] *OpenFaaS*. URL: https://www.openfaas.com/.

[26] *OpenWhisk*. URL: https://openwhisk.apache.org.

[27] 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.

[28] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. "Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider". In: *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, July 2020, pages 205–218.

[29] 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.

[30] 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.

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

[32] 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.

[33] *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/.

[34] *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/.

[35] Bartek Wydrowski, Robert Kleinberg, Stephen M. Rumble, and Aaron Archer. "Load is not what you should balance: Introducing Prequal". In: *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. Santa Clara, CA: USENIX Association, Apr. 2024, pages 1285–1299.