XX: A system

Anonymous Author(s)

Abstract

The promise of serverless currently remains an elusive and extremely attractive paradigm: get what you need as you need it, but only pay for what you use.

For example, building an elastic web server on top of a serverless infrastructure is not something that can be done easily today, but is in principle a good candidate for the serverless setting: its load is unpredictable and bursty, and it does not require managing its data locally.

Many challenges remain in making the ideal of serverless a reality; XX tackles scheduling: in a world where a large and heterogeneous set of jobs is being run on as serverless functions, there has to be a way to differentiate between functions' requirements and urgency.

XX is a distributed scheduler that allows developers to express priorities and enforces them. Developers express priorities to XX via assigning functions to fixed dollar amounts per unit of compute, and cap the overall usage by specifying a monthly budget. Memory costs per unit time used are the same across all priorities, and developers specify a maximum amount of memory for each function.[hmng: Do I need the memory blurb? I added it because I mention memory below] XX places incoming jobs on machines that have memory available, and implements a machine scheduler that enforces priorities.

[hmng: TODO impl/eval blurb]

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 also remain some things 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 lambda functions.[1] 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 end to end latencies for a simple hello world lambda function ranged from AA to BB.

What developers care about in the end is that the jobs that are important run quickly. We propose XX, a scheduling system that has *price classes* as the central metric associated with each job. Priority and preemption in the system is directly paid for through these price classes, and all of the resource allocation decisions in XX are made on the basis of class.

XX has multiple goals and challenges that it needs to achieve in order to move towards an environment where it is plausible to run web server page views entirely as serverless jobs.

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, which can then 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 100ms in scheduler queues and waiting for an execution environment before even starting to run. In a modern datacenter, where both the number of new jobs coming in and the amount of resources are extremely large, the challenge is knowing where the free and idle resources are, or finding out quickly.

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 problem is made more difficult by the fact that XX does not require memory usage limits to be given by developers, in an effort to move away from the usual bin packing with overprovisioning problem, and towards a more serverless on-demand approach also for memory. 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

This characteristic of only paying for what you use is especially attractive to developers of applications where the amount of resources that they need varies significantly over time, or is generally small, so that buying their own machines or renting a fixed amount (eg EC2) is untenable.

A central example to this paper is that of a web server. It's 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 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.

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.

However, there still are few web servers that actually run entirely on serverless offerings. There are many reasons that developers choose not to use serverless, despite in theory having workloads that are well-suited for the serverless environment. 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. A small experiment with running a lambda and measuring end to end latency shows the degree to which it varies. TODO put a histogram plot here.[hmng: how do we make it fair to aws, for example what happens if we do runs with a function that has provisioned concurrency?] What the left side of the graph tells us is that AWS is in theory capable of running jobs in a time frame that would allow a web server to serve page views from within a lambda. What the right side of the graph tells us is that the runtimes are too variable to consistently have the latency required for such a workload.

The information that AWS gets about the functions it needs to run is an amount of memory, which is then tied to a cpu power (an amount of vCPUs). 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 job latency. In fact, measurements have found that in some ways the two are inversely correlated, ie that lambdas that had more memory allocated took longer to start up.

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 they are getting when a developer puts 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 priority and costs $2\mu¢$ per cpu second, a user profile page view which is assigned a middle-high priority and cost $1.5\mu¢$ per cpu second, and finally an image processing job that can be set to a low priority which costs only $0.5\mu¢$ per cpu second.

Priorities are inherited across call chains: if a high priority job calls a low priority job, that invocation with run with high priority. 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 priorities.

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 priorities attached to jobs, which means it needs to prefer higher priority jobs over lower ones, and preempt the latter when necessary.

As shown in Figure 1, 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 priorities on the machines.

Machine Scheduler. The machine scheduler is a preemptive priority scheduler: it preempts lower priority jobs to run higher priority ones. Being unfair and starving low priority

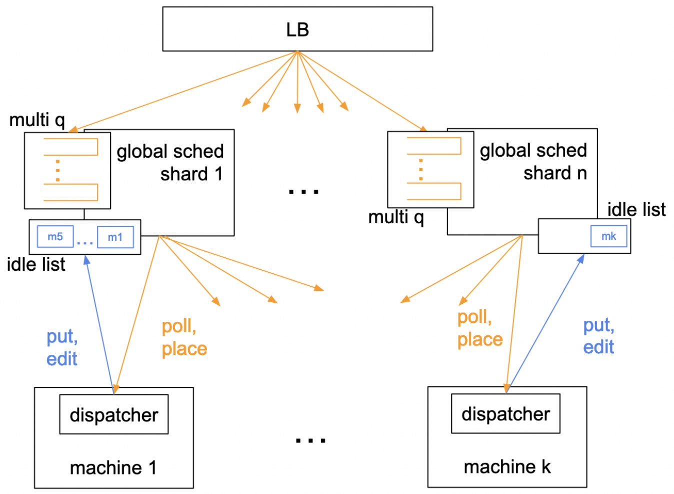

Figure 1: global scheduler shards queue and place jobs (in orange), on each machine a dispatcher thread keeps track of memory utilization and if it's low writes itself to an idle list (in blue)

jobs is desirable in XX, since image processing jobs should not interrupt a page view request processing, but vice versa is expected. Within priority classes the machine scheduler is first come first served. This matches Linux' 'sched fifo' scheduling.

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. What this means is that polling is likely to find a machine in the lower quantile of the datacenter, but not at the absolute bottom — it will not find one of the handful of machines that are actually idle. Having an idle list allows these machines, 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 priority processes quickly, without incurring the latency overheads of doing the polling to find available resources.

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 1.

The dispatcher is also in charge of managing the machine's memory. When memory pressure occurs, the dispatcher uses *priority-based swapping* to move low priority 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 priority jobs will not be run until the high priority 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 priority 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 priority. Shards choose what job to place next by looking at each job at the head of each queue, and comparing the ratio of priority to amount of time spent in the queue. This ensures that high priority jobs don't have to wait as long as lower priority jobs to be chosen next, but low priority jobs will get placed if they have waited for a while.

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. It is desirable to have a maximally heterogenous set of priorities on each machine, but since jobs come in randomly the global scheduler does not explicitly have to enforce this.

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[1], 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, priority, 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 priority* is chosen randomly, but weighted: the simulator uses a vaguely bimodal weighting across priorties. The simulator has n different priority values, each assigned to a fictitious price. Because functions are randomly assigned a priority, runtime and priority 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 and 32GB RAM each, with 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 priorities do at reflecting and enforcing SLAs. On one hand, is relevant to understand if we need priorities 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 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 how an existing state of the art research scheduler that does not take any form of priority into account performs. We simulate Hermod[1], a state-of-the-art serverless 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' priority 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 priority, as follows: deadline = runtime * maxPrice/price (as if each process were weighted by its price). This ensures that highest priority jobs' deadlines are simply their runtimes, and deadlines get more and more slack with lower priorities. 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[1].

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 2 shows the results. TODO write something once we like the graph

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, and track the memory utilization of different machines, as well as how much and what 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 3 shows the results.

5 Discussion

We have seen that having priorities helps XX deal with over-provisioning for compute. Traditionally, managing the response time for important jobs requires the overall load on the system to be low, because slowdown is averaged across

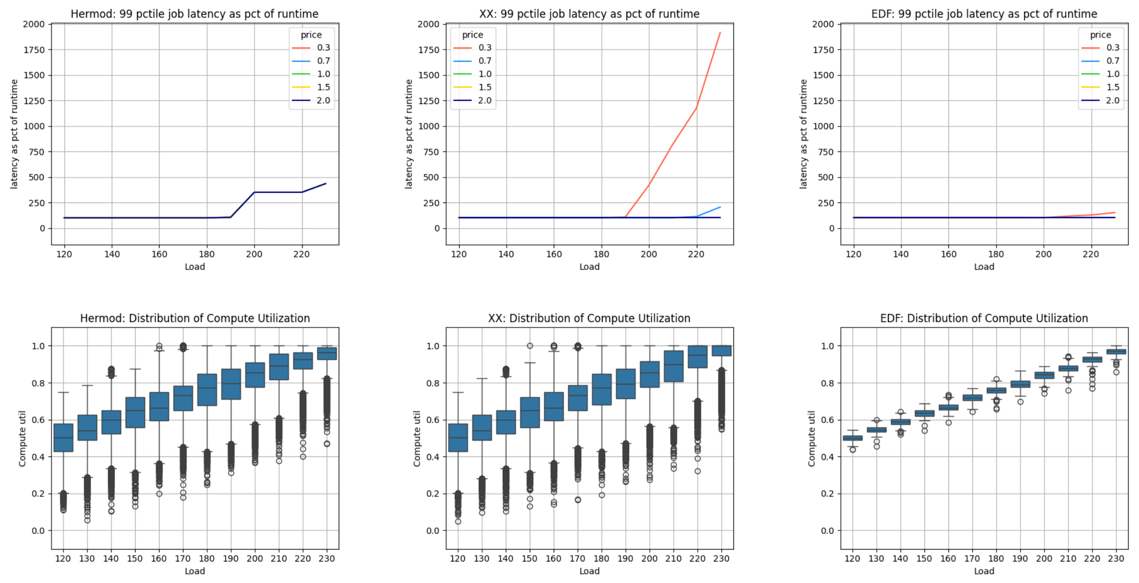

Figure 2: add caption

all the jobs that can share resources. However, having priorities allows XX to be targeted about how compute resources are shared: rather than averaging out the slowdown caused by overprovisioning across all the jobs by having them time share the cores, XX can use priorities to decide which job has to wait. This means that in priority scheduling the amount of time a job spends waiting for resources is only defined by the load of jobs with equal or higher priority.

The flipside of this is that it is possible that the entire load of the datacenter will be such that low priority jobs are starved. This is acceptable and in fact desirable for small amounts of time, but keeping this effect in check requires managing the overall load.

We propose to solve this problem by ensuring that it can never be the case the there is so much load on high priority jobs that the data center will be full with them to such a degree that other jobs can't run. There is evidence for and we expect that load on a high level will mostly be stable, with diurnal and annual patterns.[1]

This means that providers can mostly choose the rough breakdown of the load they will have at any given time (ie they can choose a percentage of how many jobs they want in each priority, and change it by adjusting prices or not allowing users to select that level anymore).

6 Related Work

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

Systems like Sparrow[1], Hermod[1], or Kairos[1] improve performance of scheduling in the distributed setting by trying out and using different scheduling policies. Unlike XX, they do not differentiate between individual types of jobs that come in.

Some systems generate information about jobs that are coming in to help placement decisions; for instance ALPS[1], which observes and learns the behaviors of existing functions and then makes scheduling decisions based on those; or Morpheus[1], which learns SLOs from historical runs, and then translates these to recurring reservations. XX instead gets the priorities directly from the developers as part of its interface.

Other existing systems ask developers to specify the desired behavior, like XX does. However, they don't change the actual underlying machine-level scheduling, and expressing priority is more complex than in XX, and/or has a different goal.[hmng: this feels a bit awk/not on the nail. Also hard because of unknown unknowns, ie there may be a related work that I don't know about for which this is not true. Hard to make specific enough so that is unlikely, while still actually going beyond just the literal design]

Sequoia[1], for instance, creates a metric of QOS for serverless functions. Unlike XX, Sequoia does not implement a new scheduler, but builds a layer in front of AWS lambda.

Another project[1] 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,

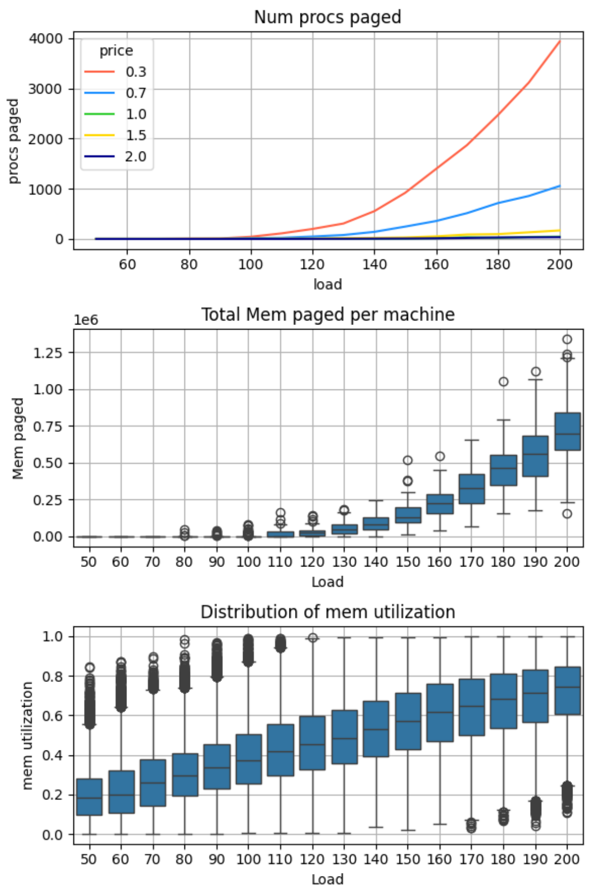

Figure 3: add caption

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[1]. 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] TODO. "TODO". In: 2020.