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 all 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.

However, for many applications it is rare to see them entirely hosted on serverless offerings today[1]. Consider a web server: its characteristics of unpredictable and bursty traffic make it well-suited for serverless, and the pay as you go model is particularly attractive to website developers who don't want to have to worry about provisioning.

Suppose a simple web server consists of two different page view handlers, one for the landing page and another for users' profile pages, and also has image processing jobs for user uploaded content. It is important that the page views finish quickly, with the landing page taking precedence over the profile pages, and the image processing jobs can be run in the background and be delayed.

If a web developer wanted to host this web server in an existing serverless offering, a popular option is AWS lambda[1]. However, AWS offers no way to prioritize different lambdas in order to ensure that the page views run quickly, while having the notion that image processing jobs don't have to. A developer using AWS lambda could implicitly prioritize page views by avoiding cold starts for only them (using reserved and provisioned concurrency[1]); or could give them more concurrency (through 'vCPUs'[1]). However, neither of these give priority to the page views when they run.[hmng: I feel like this is too vague to be a strong and convincing thing to say]

This paper's goal is to build a usable and efficient scheduling mechanism that supports priorities, in order to help enable a world of universal serverless. This goal faces multiple significant challenges.

One of the challenges 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. We focus on the aspect of scheduling latency: in a 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. For this paper, we assume that that cold start times are small enough that cold starts on the critical path are acceptable.[1]

Another challenge is that of multi-tenancy. The fact that page views are more important to the web server than image processing jobs is a relative statement: there is no way of mapping that prioritization to other developers' jobs, in order to be able to directly compare which job should run when.

A third main challenge is that of managing memory. Placing a new high priority job on a machine already running many jobs may lead to enough memory pressure to require killing an already placed job (rather than the new job just preempting existing jobs). When placing new job it is difficult to know whether the machine would have to kill something else, and if so whether it is worth it, or better to just requeue the new job.

2 Design

We propose XX, a serverless scheduler that takes into account jobs' priorities. Developers using XX write function handlers and define triggers just like they would for any existing

[1]Cold start latencies have been reduced significantly — recent state of the art systems have been able to support latencies in the single digit ms range[1].

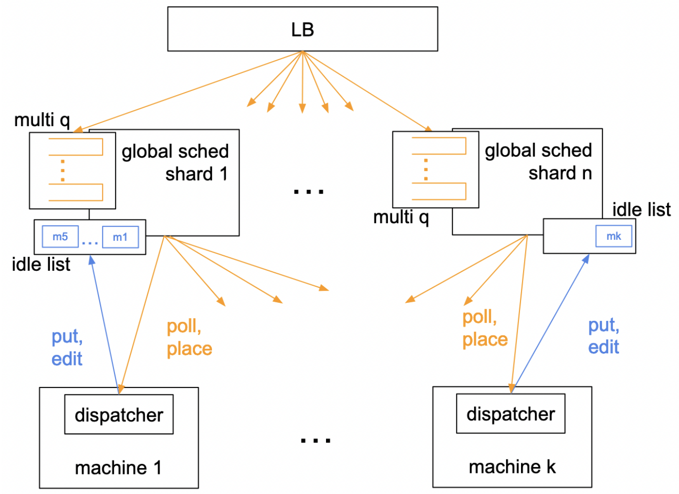

Figure 1: global scheduler shards queue and place jobs (in orange), on each machine a dispatcher thread keeps track of memory utilization and if it's low writes itself to an idle list (in blue)

serverless offering. Additionally, developers express jobs' priorities to XX, and XX enforces these priorities.

2.1 Interface

Developers express priorities to XX by assigning functions a priority value in a fixed range. This range is the same for everyone, which enables XX to directly compare different jobs' priorities. Priorities are meaningful because they are directly mapped to cost per cpu second. This ensures that developers don't simply choose the highest priority available for their most important job. For instance in the example of the web server, the home page view might be assigned a higher priority and cost 2c per cpu second, a the user profile view might be a assigned a middle-high priority and cost 1.5c per cpu second, and finally the image processing job can be set to a low priority which costs only 0.5c per cpu second.[hmng: the numbers here are way too high, but writing $0.0000002 is unfortunate]

Developers are also required to express a maximum amount of memory per function.

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.

2.2 XX Design

As shown in Figure 1, XX sits behind a load balancer, and consists of: a distributed global scheduler, which places new function invocations, a dispatcher, which runs on each machine and communicates with the global scheduler shards, and a machine scheduler, which enforces priorities on the machines.

Each global scheduler shard maintains 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 some information about compute pressure. This information allows the global scheduler to place high priority processes quickly, without incurring the latency overheads of finding available resources.

Global scheduler shards store the jobs waiting to be placed in a multi queue, with one queue per priority.

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 jobs is desirable in XX, since image processing jobs should not interrupt a page view request processing, but vice versa is expected.

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 also responds to probes by shards: given a potential job that a shard might want to run on the machine, the dispatcher computes the *time to profit*, which is the time it would take for the machine to start making a profit off of the decision of placing the job there. If there is enough memory free to fit the new job's max memory, that number is 0. If the dispatcher might have to kill a process due to memory pressure, it computes which job it would kill, which is the job with minimal price where $j.memUsg > newJ.maxMem$. The time to profit then is $(jToKill.timeRun \cdot jToKill.price)/(newJ.price - jToKill.price)$.

The dispatcher is also in charge of killing jobs under memory pressure, should it occur. It chooses the job to kill by looking at both memory used and money wasted if killed (lower priority jobs should be the ones to be killed if possible, but won't help much if they weren't using any memory to begin with). The dispatcher requeues killed jobs at a randomly chosen shard.

Global Scheduler Shards.

Procedure 1 Choosing a machine for a job j

```
$N$ = { machines in freeList with memAvail > j.maxMem }
**if** $|N| > 0$ **then**
**return** min(N.maxPriorityRunning, N.qSize)
**end if**
$M$ = timeToProfit of k polled machines
**if** min(M.timeToProfit) < $THRESH$ **then**
**return** min($M$)
**else**
    reQ j, with priority -= 1
**end if**
```

Shards choose what job to place next by looking at each job at the head of a queue in the shards multi-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 finds a machine to run it, shown in Procedure 1. The shard will first look in its idle list for a machine that has the job's maximum memory available. If there are multiple such machines, the shard looks at each machines compute pressure metrics; the goal being to minimize cpu idleness and job latency in low load settings.[hmng: playing with deatils of this in the scheduler right now]

If a machine from the idle list is chosen, the response from the dispatcher upon placing the job will include updated utilization information, which the shard will use to update the idle list entry.

If there are no machines in the idle list with the memory available, the shard switches over to power-of-k-choices: it polls k machines, sending the price and the maximum memory of the job currently being placed, and getting back the time to profit. The shard then can choose to place the new job on the machine with the minimum value, or if all of them are too high the shard re-queues the job.

3 Preliminary Results

In order to evaluate XX, we explore the following questions:

(1) Are priorities necessary?
(2) Are priorities sufficient to describe the desired behavior?
(3) Is XX good at enforcing priorities?

To explore these questions, we built a simulator in go[1], which simulates different scheduling approaches.

3.1 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. 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 uniform random between 1MB and 10GB.

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) functions use all of their memory right away
(3) communication latencies are not simulated

3.2 Are priorities necessary?

To answer the first question, we explore how an existing state of the art research scheduler that does not take any form of priority into account performs on the web server's workload. We simulate Hermod[1], a state-of-the-art serverless scheduler, and compare it to a simulation of XX.[hmng: is XX really the right thing to compare to? or should it be something more minimal but with priorities] Hermod is a scheduler built specifically for serverless, and 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.[hmng: do we talk about the nuance with different load balancing techniques in low vs high load?] Hermod does not use priorities in its design, and as such the simulator ignores jobs' priority when simulating Hermod's design.

We simulate both designs, and compare the latencies that each achieves, primarily for the page view jobs. A strong result for XX would show that latencies for higher priority jobs start going up at a higher load than for Hermod, which would indicate that Hermod's scheduling cannot perform as well in high load settings on high priority jobs. Figure 2 shows the results. We can see that as expected, Hermod's slowdown, even in middle load settings, extends to the high priority jobs, while in XX the low priority jobs are the only ones that are affected.[hmng: ok but how is this not obvious: we made a new metric and look we did better at it than other systems that did not use that metric wow what a surprise]

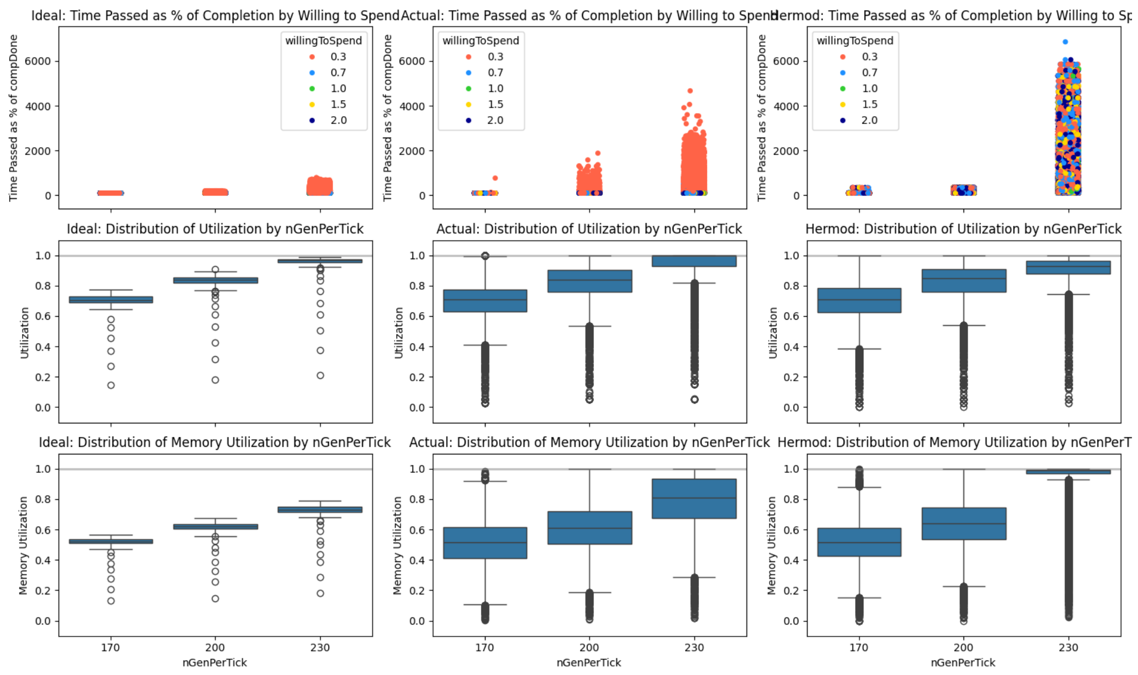

Figure 2: a placeholder graph

3.3 Are priorities sufficient?

Answering the second question requires defining the desired behavior, which comes down to job latency. So, we assign each job invocation a deadline, and to make things simpler we allow the simulator perfect knowledge of the job's runtime. We can then define desired behavior as meeting all of these deadlines, provided such a thing is possible. Simply setting the deadline to be the runtime is not realistic: perhaps for high priority jobs that is the case, but for low priority jobs that is not true, the reason they are low priority is because they are willing to wait. We thus define the deadline as a function of the runtime as well as the priority, where deadline = runtime * 1/priority (as if each process were weighted by its priority).[hmng: check this math does what we want it to] We then simulate an Earliest Deadline First (EDF) scheduler, which is queueing 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 observerd in this EDF simulated scheduler with an idealized version of a preemptive priority scheduler, and look at the latencies they both get. Because this is purely about core scheduling, both ignore memory in this simulation. A good result for XX would show little difference between the two: we expect the EDF version to do better, because it is both theoretically optimal and has access to perfect information, but if the priority scheduler does similarly then we know that it can be a good proxy for deadlines. We can see the results in Figure 2.[hmng: TODO do this]

3.4 Is XX good at enforcing priorities?

To answer the third question, we take an idealized version of the scheduling XX does, and compare XX's performance to it. The idealized scheduler runs in a centralized setting: it is as if the whole datacenter is one machine with many many cores. As such there is no memory fragmentation, and its utilization represents an optimal solution.[hmng: this feels slightly silly as an experiment, because data points don't give a context to be able to tell how close is good. Maybe do an ablation study vibe, where we look at our techniques and see which ones help?]

We compare the latencies as well as the memory and compute utilization we observe. A good result for XX would show that the two do not differ too much[hmng: super ill defined]; we expect utilization to have a higher variance for the XX version of the simulator, but a good result would show that we are able to reduce the variance and raise the average and tail utilization by using idle lists. We can see the results in Figure 2.[hmng: TODO do this]

4 Discussion

In this section we discuss the the way that having priorities would impact how we can approach some of the open questions that remain about how a scheduler should best manage resources.

4.1 Managing memory

In section 2, XX makes decisions based on maximum memory usage estimates for each job, provided by the developer. However, in reality, making scheduling decisions based off of this sort of estimate of a maximum can lead to underutilization. Jobs often have highly variable memory usage; for instance the distribution of content among users of social media platforms is notoriously skewed[1], so jobs that fetch information for a user and process it will be run over very different amounts of data depending on who the user is. So making scheduling decisions based off of an estimate of a maximum will lead to memory underutilization, and instead we are forced to play a game of overprovisioning.

Having priorities can help. We overprovision in order to achieve high utilization, so need a way of dealing with a situation where there are not enough resources for all the jobs on a machine. Under that situation of high memory pressure, we suggest a priority-based paging approach, where the lower priority jobs' memory are paged out. Because machines run highest priority first scheduling, we know that the paged job will not be run until the higher priority jobs have finished (and thus freed their memory).[hmng: there's a caveat here where they could block on i/o, but the likelihood that all the jobs except the last one block on i/o is low] Later, when there is lower load, and the previously paged job is ready to run again, the machine can page back in all of its memory. Thus the paged job pays no latency overhead for having been paged, and the amount of time spent paging memory is low: once to move all the memory out, and once to move it back in. This is the minimal amount of data movement necessary to free up the required memory without deleting/killing.

4.2 Managing compute

We have already seen that having priorities helps XX deal with overprovisioning on the compute side. Traditionally, managing the response time for important jobs requires the overall load on the system to be low, because slowdown is averaged across 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 sllowing users to select that level anymore).

[hmng: We discuss what a good breakdown would look like in the next section? Put eval next? Or just don't discuss and leave it at that, although that seems a little vague.]

5 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 betwen 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, rather than assocaiting 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.