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 views, one for the landing page and another for users' profile pages, and also has map reduce jobs for data processing. It is important that the page views finish quickly, with the landing page taking precedence over the profile pages, and the map reduce jobs can be run in the background and be delayed.

If a web developer wanted to host their web in an existing serverless offering, a popular option is AWS lambda[1]. However, AWS offers no way to distinguish between different lambdas in order to ensure that the page views run quickly, while having the notion that map reduce 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); or could give them more concurrency (through 'vCPUs'). 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 focuses on building a usable and efficient scheduling mechanism to support differentiating between jobs of different priorities in a world of universal serverless. In doing so, it faces multiple significant challenges.

One of the challenges is that of placing jobs quickly enough. This means that a job that takes 20ms to run cannot spend 100ms in scheduler queues and waiting for an execution environment before even starting to run. One part of this challenge is cold starts, but for this paper, we assume that that generally cold start times are small enough compared to the jobs being run that cold starts on the critical path are acceptable.[1] 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.

Another challenge is that of multi-tenancy. The fact that page views are more important to a web server than map reduce jobs is a relative statement: there is no way of mapping that prioritization to other peoples jobs, in order to be able to directly compare which job should run when. And, in fact, developers cannot be trusted to make such a comparison, their own jobs will always seem more important than others': if given an absolute scale from 0-99 (0 being the highest priority), the highest priority job will always get a 0 and the rest will be relative to that. However, that does not reflect that in fact different clients do have different requirements and expectations, and need to be treated differently.

A third main challenge is that of managing memory. Given that machines are limited in memory, placing a new high priority job invocation on a machine already running many jobs

[1] Cold start latencies are a popular area of research and as a result have been reduced significantly — recent state of the art systems have been able to support latencies in the single digit ms range[1]. We expect this trend to continue.

may or may not lead to enough memory pressure to require killing an already placed job (rather than just preempting other jobs). It is difficult to know whether that would be the case, and if so whether it is worth it, or better to just requeue the new job.

2 Design

Developers using XX write function handlers and define triggers just like they would for any existing serverless offering. Additionally, developers express jobs' priorities to XX, and XX enforces these priorities.

2.1 Interface

Developers express priorities to XX via assigning functions to fixed priorities that map directly to dollar amounts per unit compute. For instance in the example of the web server, the home page view might be assigned a high 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 map reduce job can be set to a low priority which costs only 0.5c per cpu second. This enables the priorities to be globally comparable, while disincentivizing developers to always choose the highest possible priority for their most important job.

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.

Finally, developers are required to express a maximum amount of memory per function.

2.2 XX Design

XX 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.

Attached to each global scheduler shard is 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 allows the global scheduler to place high priority processes without incurring the latency overheads of finding available resources.

The global scheduler stores the jobs waiting to be placed in a multi queue, with one queue per priority.

An overview of XX's structure can be seen in Figure 1.

Machine Scheduler. The machine scheduler is simple: jobs have fixed priorities, and higher priorities preempt lower priorities. Being unfair and starving low priority jobs is desirable, since map reduce jobs should not interrupt a page view request processing, but vice versa is expected.

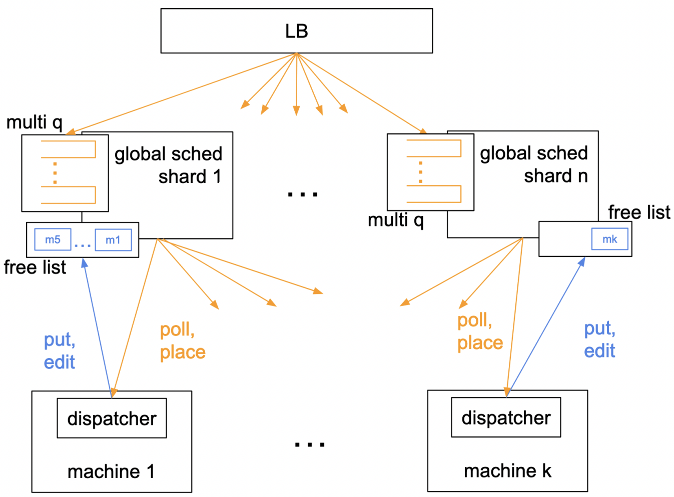

Figure 1: global scheduler shards queue and place jobs (in orange), on each machine a dispatcher thread keeps track of memory utilization and if it's low writes itself to a free list (in blue)

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,

Algorithm 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
```

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 choose what job to place next by the ratio of priority to amount of time spent in the queue, so high priority jobs don't have to wait as long as lower priority jobs to be chosen next.

When placing a job, the shard finds a machine to run the it, as also shown in the pseudocode in Algorithm 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, this time with a lower priority.

Checking in with clients' budgets to ensure that usage is not significantly outpacing a rate compatible with the budget is done asynchronously: each time a job for a client is triggered a global counter is asynchronously updated. If the counter's rate of change increases absurdly with regards to previous behavior as well as the budget as a whole, this triggers a throttling for that job.

3 Preliminary Results

To understand the dyanmics of XX, we built a simulator in go[1]. Our simulator runs three different worlds: an ideal scheduling world, an implementation of XX's design, and an implementation of Hermod[1].

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. It still has the same limited memory as the datacenter though, so will kill using the same mechanism as XX.[hmng: not sure if this is clear and I should just shut up or if it's non-obvious and I should explain it more]

Hermod is a scheduler built specifically for serverless, and is the result of a from-first-principles analysis of different scheduling paradigms through a simulator. The paper simulates different scheduling approaches along three dimensions: late-vs early binding; Processor Sharing (PS) vs First Come First Served (FCFS) vs (idealized) Shortest Remaining Processing Time (SRPT); and least loaded vs locality based vs random load balancing. The paper concludes that early binding is best, in combination with PS (PS is close enough to SRPT with perfect knowledge, which is impractical in reality). For the load balancing Hermod uses a hybrid mechanism: at low load, Hermod packs machines one by one, each up until the number of cores (a water-filling like mechanism, where each machine is filled until each core is running a job); and at high load it chooses the least loaded machine. Hermod does not use priorities in its design, and as such we ignore jobs' priority when placing them in the Hermod world in our simulator.

In 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 maximum 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 five different priority values, each assigned to the prices of 3c, 7c, 10c, 15c, 20c.[hmng: does it matter that these prices are unrealistic] Because functions are randomly assigned a priority, runtime and priority are not correlated. *Maximum memory* is chosen uniform random between 1MB and 10GB.

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

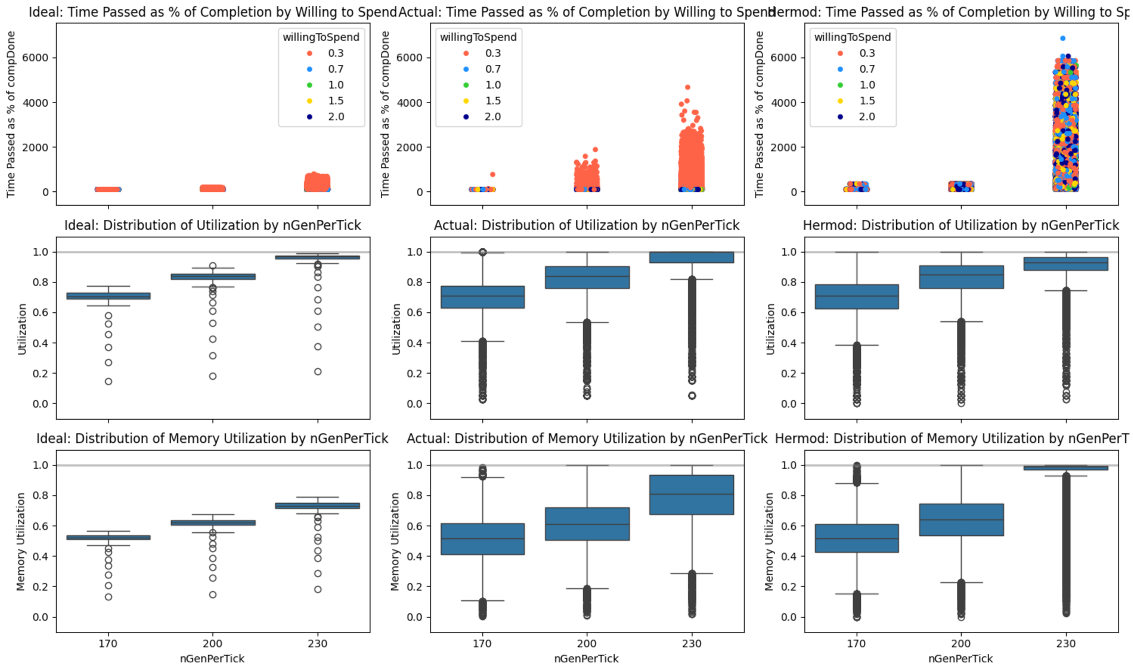

Figure 2: a first version of the graph

We then run the simulator at different load points, and track latencies as a percentage of the required runtime, as well as memory and compute utilization; for each of the three different worlds of XX, Hermod, and ideal.

A good result for XX would be doing in between Hermod and the ideal world. We expect XX will do better than Hermod because it has more knowledge: whereas Hermod will spread latency hits across all jobs indiscriminately as load goes up, XX can keep high priority jobs running quickly and limit the slowdown to only affect lower priority jobs. We cannot expect XX to do as well as the ideal world: not only does XX have to deal with memory fragmentation, which the ideal world does not, but it also has imperfect information — once the idle list is empty, XX relies on k choices to find a fitting machine.

We see the results of a run in Figure 2. As expected, XX does not perform as well as the ideal world: latencies rise more quickly, and both compute and memory utilization have a higher variance across machines and time. We can also see that, as expected, under higher load XX is able to keep all priorities except the lowest running with low latency, whereas Hermod slows down all the functions the same.

4 Discussion

4.1 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 we are forced to either get better estimates, or play a game of overprovisioning, because reserving the maximum amount of memory would lead to, in most cases, low memory utilization.[hmng: who is 'we' now? it's not XX, but it's also not some other concrete system, it's kinda the hypothetical one we're discussing]

One option is profiling: as ML models improve, so our ability to predict things improves, including predicting memory usage. Especially for jobs with a high variance of memory usage, using a model that is given the inputs to invocations could work well, since the inputs are likely to be the determining factor for memory usage (for instance, the model could learn which users are the ones that have huge amounts of data). However, predictions are still just estimates that could always be wrong, and the system needs to be able to handle that. They also need to be based on profiling information, which will be slow to adapt to changes in code and in the underlying data.

Another option is snapshotting: lower priority functions are allowed to snapshot themselves and then are placed somewhere else and re-started. This way none of their already done computation is wasted, and the memory is freed. However, the timescale would have to be such that the snapshotting does not (in its use of memory or compute) prohibitively block the other jobs on the machine from running. This works against the incentives of the situation: ideally we want to free a large amount of memory, but the time it takes to snapshot a job scales with the amount of memory it is using.

A stronger option is priority-based paging: the lower priority processes' 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 processes 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 processes 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 Compute

XX uses preemptive priority scheduling on the machine scheduler level. This means that at any moment in time, the process currently running is the one with the highest priority on the machine. This ensures that the web server's page views run with high performance, and the map reduce jobs do not interrupt but rather wait for lulls in load to run. The flipside of this is that if there are no such lulls, ie if there are so many high priority jobs that they take up all the resources all the time, it is possible that nothing of any lower priority would ever run at all.

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 will mostly be stable: this supported by the strong law of large numbers (it is very unlikely that all the jobs' random bursts align), and can be seen in the ways AWS prices things: spot instances are sold at a market rate determined by the amount of resources available in a given zone[1], and the market rate is experientially very steady over time[1]. The prices for lambdas and ec2 instances also only changes very slowly[1].[hmng: Hermod had a good way of putting this slightly more concisely and cleanly, look at what they did]

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.