Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Systems@ETH Zürich

Cloud Computing Architecture

Semester project report

Group 018
Aristotelis Koutris - 23942683
Georgios Manos - 23948342
Maria Tsanta - 22952857

Systems Group
Department of Computer Science
ETH Zurich
May 17, 2024

Part 3 [34 points]

1. **[17 points]** With your scheduling policy, run the entire workflow **3 separate times**. For each run, measure the execution time of each batch job, as well as the latency outputs of memcached, running with a steady client load of 30K QPS. For each batch application, compute the mean and standard deviation of the execution time [1] across three runs. Also, compute the mean and standard deviation of the total time to complete all jobs - the makespan of all jobs. Fill in the table below. Finally, compute the SLO violation ratio for memcached for the three runs; the number of data points with 95th percentile latency $>$ 1ms, as a fraction of the total number of data points. The SLO violation ratio should be calculated during the time from when the first batch-job-container starts running to when the last batch-job-container stops running.

job name	mean time [s]	std [s]
blackscholes	112.67	2.49
canneal	132.67	0.47
dedup	23.33	0.47
ferret	141.33	0.47
freqmine	102.67	0.47
radix	9.67	0.47
vips	31.67	0.47
total time	152.33	0.47

Our scheduling policy managed to fulfill the SLO goals completely, resulting in no SLO violations in any of our runs.

Answer:

Create 3 bar plots (one for each run) of memcached p95 latency (y-axis) over time (x-axis), with annotations showing when each batch job started and ended, also indicating the machine each of them is running on. Using the augmented version of mcperf, you get two additional columns in the output: `ts_start` and `ts_end`. Use them to determine the width of each bar in the bar plot, while the height should represent the p95 latency. Align the x axis so that $x = 0$ coincides with the starting time of the first container. Use the colors proposed in this template (you can find them in `main.tex`). For example, use the vips color to annotate when vips started and stopped, the blackscholes color to annotate when blackscholes started and stopped etc.

Plots:
We split the data from each run into two plots. One displays p95 latency over time and the other displays the jobs running at each moment. The time axes for both plots is aligned so that $x = 0$ coincides with the starting time of the first container.

[1] Here, you should only consider the runtime, excluding time spans during which the container is paused.

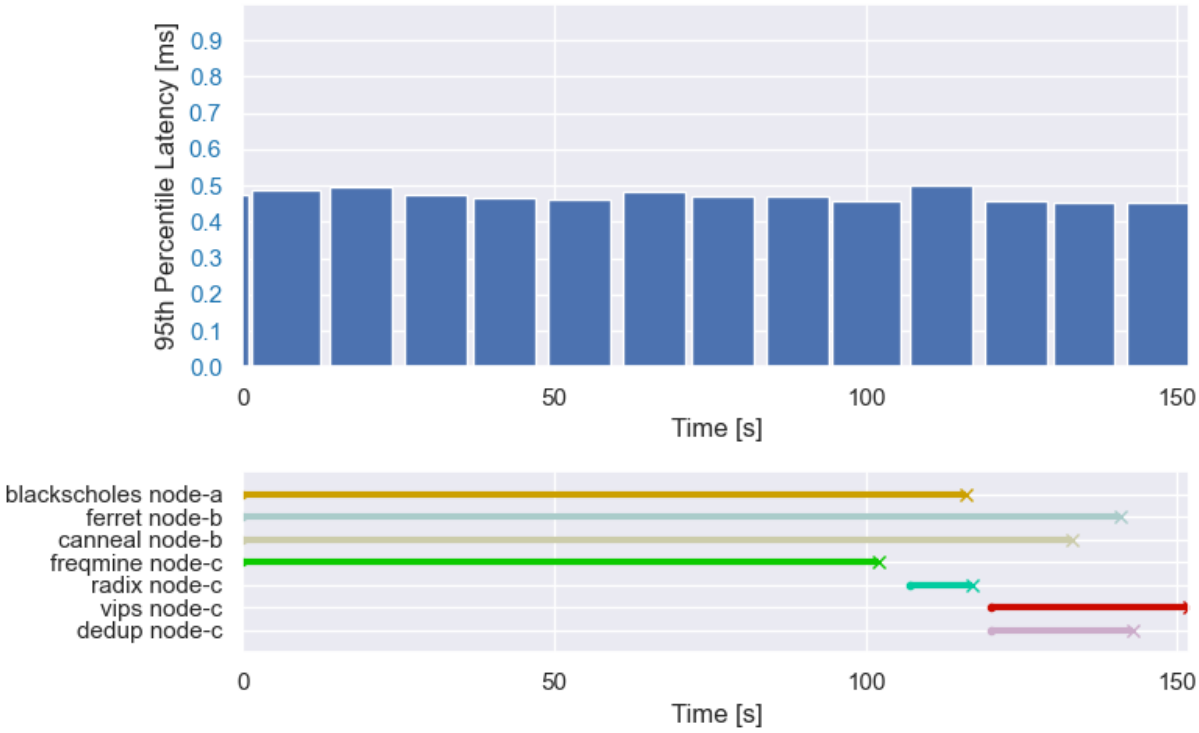

Figure 1: First run.

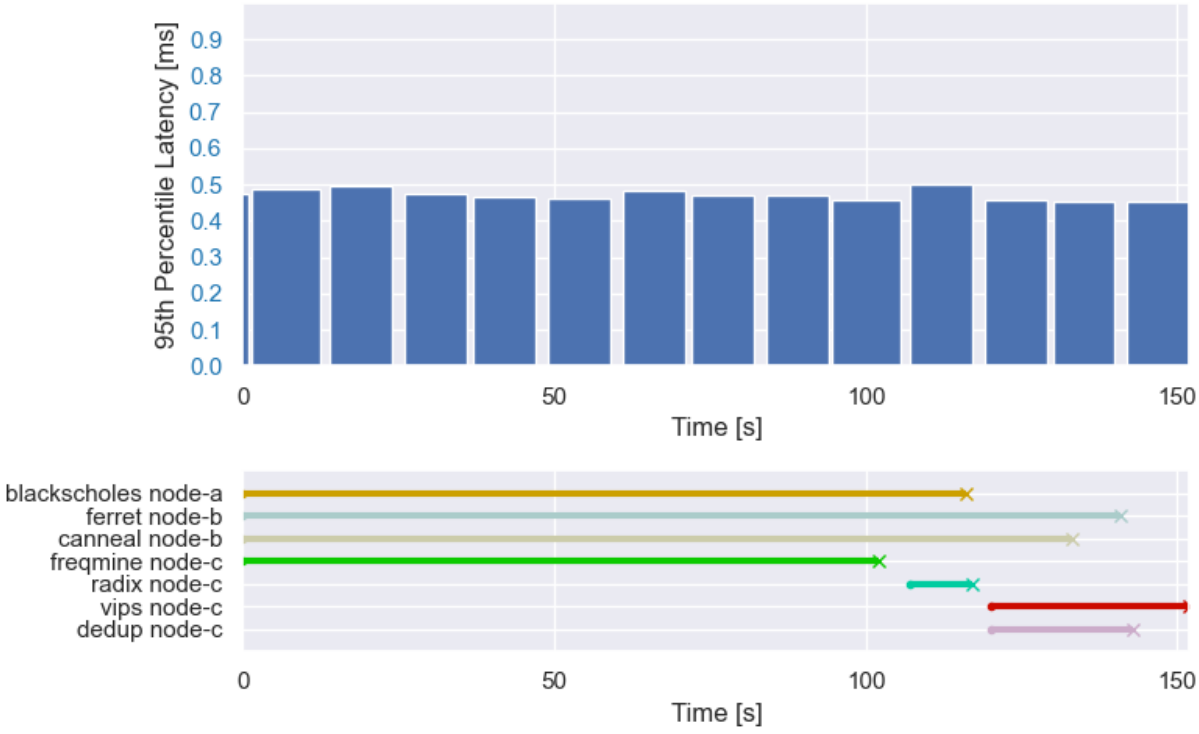

Figure 2: Second run.

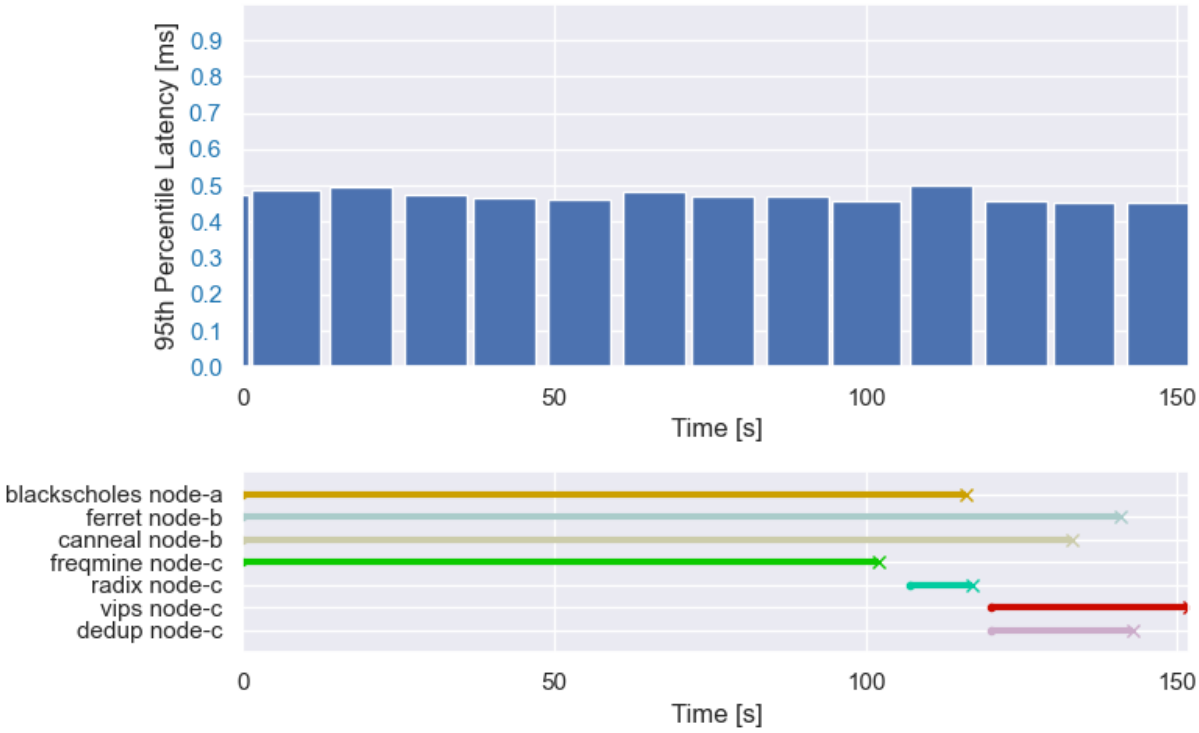

Figure 3: Third run.

2. [**17 points**] Describe and justify the "optimal" scheduling policy you have designed.

   - Which node does memcached run on? Why?

     Answer:
     From our previous analysis of `memcached` we know that it is the most sensitive to CPU and L1 instruction cache interferences, which means that those are the resources it relies on the most. For those reasons we placed `memcached` on the 1st core of `node-a-2core`, which is of type `n2d-highcpu-2`. This node only has 2GB of RAM, but can efficiently handle `memcached` since it requires CPU power but at the same time has low high memory demands. It is a safe choice as most of our jobs benefit a lot from parallelization and thus could utilize 2 or more cores of a CPU. Also at 30k QPS, we saw that on a normal CPU, memcached's 95th percentile latency without any interference was $\approx$ 1ms, and thus our SLO would be more susceptible to violations if we used such a cpu.

   - Which node does each of the 7 batch jobs run on? Why?
     To choose where each batch job is run we first studied the characteristics of our nodes. We found that `n2d-highcpu-2` (`node-a-2core`) has 2 cores that offer high performance and are optimized for CPU intensive tasks (e.g. batch processing or scientific calculations). `n2d-highmem-2` (`node-b-4core`) has 4 CPUs and 32 GBs of memory, supporting a larger amount of memory per vCPU. Finally `e2-standard-8` (`node-c-48core`) has 8 CPUs and 32 GBs of memory, which is better suited for multithreaded workloads. To lock jobs in specific nodes, we use the pod's `nodeSelector` property in our `yaml` file. This way the pod is restricted to use the label we specify and cannot run on a node other than the one

we have specified. We also used `taskset` to mount some jobs on specific cores of a node, and `resources.requests` and `resources.limits` for CPU and memory requirements.

Answer:

- blackscholes: this batch job runs on the `node-a-2core` node. The program is limited by the amount of floating-point calculations a processor can perform [1], thus can benefit from a CPU with higher performance. Also, doesn't have high memory requirements so it can efficiently run in the node with the least RAM available.
- canneal: this batch job is running on the `node-b-4core` node. Since `canneal` is not sensitive to the interference on most resources, we chose to run it here as its `native` working set has a size of 2 GBs [1] and thus cannot fit on `node-a-2core`
- dedup: this batch job is running on `node-c-8core`. `dedup` also has a `native` working set of 2GBs [1] and thus cannot fit in `node-a-2core`. At the same time, it benefits little when working with more than 2 threads, is not sensitive to memory bandwidth interference, and thus this node is a suitable to run in parallel with another job.
- ferret: this batch job is running on the `node-b-4core` node. It is highly sensitive to memory bandwidth interference, it benefits a lot from multithreading and the speedup going from 4 to 8 threads is insignificant, thus making this node ideal for this job.
- freqmine: this batch job is running on the `node-c-8core` node. It has a `native` working set of 1 GB [1] and thus it wouldn't be risky to place it on `node-a-2core`, but more importantly it benefits a lot from multithreading. Therefore, the job runtime can be reduced significantly by running on 8 cores.
- radix: this batch job is running on the `node-c-8core`. It is insensitive to most Sources of Interference (SoI), but highly benefits from multithreading on 8 threads. Therefore this node is a really good option to minimize the job's running time.
- vips: this batch job is running on the `node-c-8core`. The job is sensitive to all types of interference but benefits significantly from multithreading up to 8 threads, making this node a good option for this job.

- Which jobs run concurrently / are colocated? Why?

  Answer:

  - `memcached` and blackscholes are executed concurrently in `node-a-2core`, taking advantage of the fast CPU as both jobs benefit significantly from a faster CPU. They also have small working loads that can fit in a 2GB memory and do not benefit significantly from multithreading, and blackscholes is mostly sensitive to memory and llc, while `memcached` is sensitive to CPU interference.
  - ferret and canneal are executed concurrently in `node-b-4core` since canneal is sensitive only to llc, while ferret is sensitive to memory bandwidth, llc and CPU, so they could both benefit from the high memory bandwidth available while also utilizing from 2 to 4 threads. Note that these 2 jobs are a necessary bottleneck for our runs, but the reasoning of this choice is explained below.
  - freqmine and radix run sequentially before vips and dedup are executed in `node-c-8core`. While dedup can benefit significantly for up to 2 threads, the rest of the jobs benefit significantly for 4 or more threads. Therefore dedup can be executed using up to 2 cores, leaving the rest for vips. Also, dedup has a large working set so it cannot fit in

`node-a-2core` after blackscholes, and `node-b-4core` finishes its jobs a few seconds after radix is done.

- In which order did you run the 7 batch jobs? Why?

  Order (sorted): blackscholes - ferret - canneal - freqmine, radix, vips - dedup (dash means jobs start concurently)

  Why:

  By our findings in the previous part[2], the jobs that take the most time are (in order, based on their 8 threads performance without interference) 1. freqmine, 2. canneal, 3. ferret, 4. blackscholes. Ideally, to minimize execution time, we would run these time consuming jobs in separate nodes so they do not interfere with one another, and schedule the rest of the jobs to run in parallel. However, two of these jobs, namely freqmine and canneal, can't run on `node-a-2core` due to increased memory requirements. Therefore, it is necessary to collocate two of those jobs on the same node. To minimize the impact of this, we chose to run ferret with canneal on `node-b-4core` , freqmine on `node-c-8core` as it benefits a lot from fully using all 8 cores, and finally blackscholes on `node-a-2core` as it benefits significantly from the fast CPU. This way, freqmine takes approximately 01:43 to complete while blackscholes takes 01:56, and canneal, ferret take approximately 02:23 to finish while running concurrently.

  The final issue to address was where to run the rest of the jobs. radix running on 8 cores takes about 10 seconds to complete, and thus can run after freqmine to finish as early as possible (also does not fit on `node-a-2core`), and vips and dedup can be executed in parallel as they can share the 8 cores (6 + 2 respectively) and take about 30 seconds to complete, so the time needed for all jobs in `node-c-8core` is a few seconds over `node-b-4core`.

- How many threads have you used for each of the 7 batch jobs? Why?

  In general, it is a good idea to have as many threads as the number of available cores [1], or close to that number (always `num_cores` $\leq$ `num_threads`), to allow most sytems to fully utilize multithreaded parallelization, while not adding too much overhead. Some jobs are pinned to specific cores, while others are not as they run in parallel with other jobs and could benefit from the available cores as soon as the first job is done executing.

  Answer:

  - blackscholes: 2 threads while it runs on 1 core. Based on our previous analysis we saw that `blackscholes` does not achieve significant speedup from multithreading, but we know that its performance is limited by the number of flops the CPU may perform [1], so it is best to run on the 2nd core of the fast CPU and have only 2 threads.
  - canneal: 4 threads, running on 2 to 4 cores. Based on our previous analysis we saw that while `canneal` had a bigger speedup for more than 4 threads the overall time of the job was still ≈ 100 seconds so it would not have been effective to increase the number of threads further.
  - dedup: 2 threads running on 2 cores. Based on our previous analysis we saw that `dedup`'s speedup was pretty much the same after we reached 2 threads so there was no reason to dedicated more cores here.

[2] even though many of these jobs were executed with `simlarge` set instead of `native`, we still used their execution time as a soft lower bound

- ferret: 4 threads running on 2 to 4 cores. Based on our previous analysis we saw that `ferret`'s speedup was significant for 4 threads, but small speedup after that.
- freqmine: 8 threads running on 8 cores. Based on our previous analysis we saw that `freqmine` achieved a very good speedup of 4x on a high number of threads, that could not be reached with a smaller thread count.
- radix: 8 threads running on 8 cores. Based on our previous analysis we saw that `radix` achieved a very good speedup of 6x on a high number of threads, that could not be reached with a smaller thread count.
- vips: 6 threads running on 6 cores. Based on our previous analysis we saw that `vips` achieved a very good speedup of 4x on 4-8 threads, but the speedup from 4 to 8 was small so there wasn't much reason to push for 8 threads.

- Which files did you modify or add and in what way? Which Kubernetes features did you use?

  Answer:
  We have modified the `parsec-benchmarks yaml` files to set up the number of cores, applied resource limits (CPU, memory) and set the number of threads we want on each job, as well as our node preference for them. We have changed the `memcache-t1-cpuset yaml` file for that reason as well. To also avoid pulling the docker images every time, we changed the pull policy from `Always` to `IfNotPresent`, so only on the first run the server actually pulls the docker images and caches them for the next runs.
  We have added scripts to deploy the cluster (`deploy_cluster.sh`), setup the memcached service (`setup_memcached.sh`), run the benchmarks (`run_benchmark.sh`, `run_parsec.sh`) and then finally delete our cluster (`delete_cluster.sh`)[3]. Note that `run_benchmark.sh` also kills memcached agents and client measure and runs them again to get correct measurements on our scheduling policy.
  Apart from the standard Kubernetes commands, we also use the following command `kubectl get pods -o json > results.json` to get the results from the jobs so that we could run the timing script provided. We also tried to setup a Daemonset to pull docker images so the jobs are not delayed by them without much success, but this difference is mitigated after running the `run_benchmarks.sh` script once as on the first run it will download and cache the docker images (since we changed the `imagePullPolicy` as mentioned above), significantly reducing the total tail time on the next runs.

- Describe the design choices, ideas and trade-offs you took into account while creating your scheduler (if not already mentioned above):

  Answer:
  Our design choices and ideas are thoroughly explained above. To summarize, the general idea was to prioritize efficiently running longer jobs and to allocate resources such as cores and memory based on each job's characteristics (e.g. parallelization speedup, working set). Additionally, we aimed to minimize node idle time, which entails that nodes should finish their jobs approximately at the same time. Finally, jobs which were collocated where chosen so that they are sensitive to pairwise exclusive SoIs. Of course, some resources are exclusive to each core (L1/L2 caches, CPU) while others are shared across cores (LLC, memory bandwidth and size), therefore running 2 jobs on the same node we expect to have some slowdown as there will be interference on needed resources.

[3] The scripts should be run in order of deploy - setup `memcached` - run benchmarks - delete cluster

To get the absolute fastest time possible, we do have to try and squeeze more jobs into `node-a-2core` or experiment with sub-core resources (i.e. allocate 50% of a CPU on a job), however the number of good candidates is limited (canneal, dedup, freqmine, radix do not fit, but also ferret is a long job and both it and vips benefit a lot from multithreading) and for the sake of our resources did not attempt to design a more complicated policy.

Please attach your modified/added YAML files, run scripts, experiment outputs and the report as a zip file. You can find more detailed instructions about the submission in the project description file.

Important: The search space of all possible policies is exponential and you do not have enough credits to run all of them. We do not ask you to find the policy that minimizes the total running time, but rather to design a policy that has a reasonable running time, does not violate the SLO, and takes into account the characteristics of the first two parts of the project.

Part 4 [74 points]

1. **[18 points]** Use the following `mcperf` command to vary QPS from 5K to 125K in order to answer the following questions:

```
$ ./mcperf -s INTERNAL_MEMCACHED_IP --loadonly
$ ./mcperf -s INTERNAL_MEMCACHED_IP -a INTERNAL_AGENT_IP  \
           --noload -T 16 -C 4 -D 4 -Q 1000 -c 4 -t 5 \
           --scan 5000:125000:5000
```

   a) [7 points] How does memcached performance vary with the number of threads (T) and number of cores (C) allocated to the job? In a single graph, plot the 95th percentile latency (y-axis) vs. achieved QPS (x-axis) of memcached (running alone, with no other jobs collocated on the server) for the following configurations (one line each):

   - Memcached with T=1 thread, C=1 core
   - Memcached with T=1 thread, C=2 cores
   - Memcached with T=2 threads, C=1 core
   - Memcached with T=2 threads, C=2 cores

   Label the axes in your plot. State how many runs you averaged across (we recommend three runs) and include error bars. The readability of your plot will be part of your grade.

   Plots:

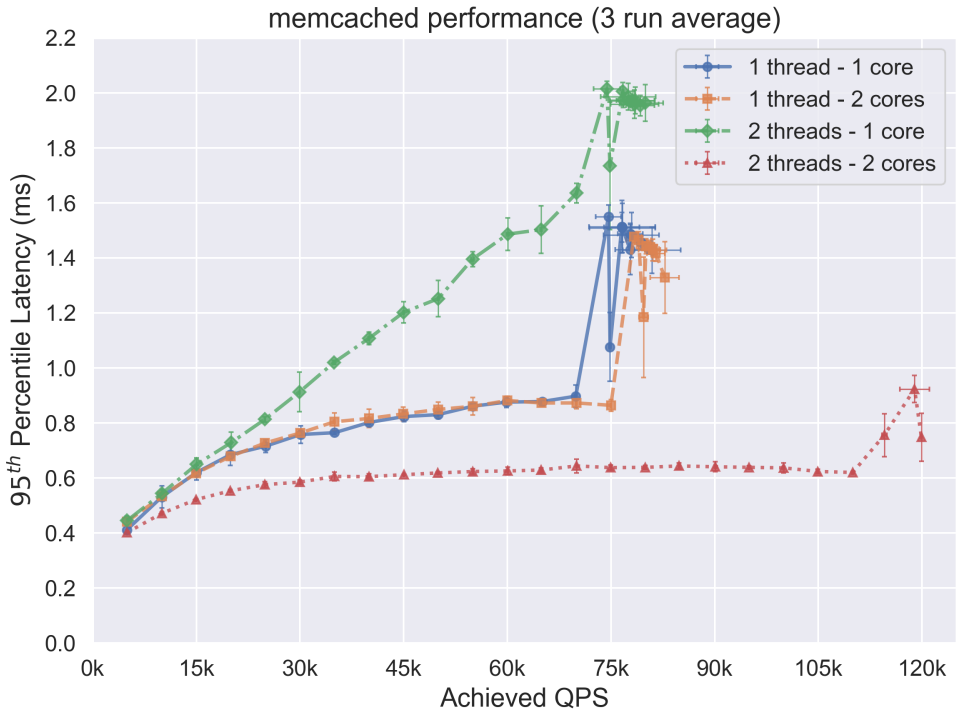

Figure 4: `memcached` 95th percentile latency for achieved QPS values, averaged across three runs, with standard deviation error lines accross latency and achieved QPS.

What do you conclude from the results in your plot? Summarize in 2-3 brief sentences how memcached performance varies with the number of threads and cores.

Summary:
We know that memcached is sensitive to CPU SoI, but also, as memcached can serve requests from multiple input threads [2], it will also benefit from having more cores available to distribute the load.
This explains why using 2 threads in 1 core gives us the worst performance since both threads will be requesting CPU resources from the same core, but also running 1 thread in 2 cores makes small improvement on high loads as memcached is not able to fully utilize the 2 available cores. Running 2 threads for 2 cores is the best configuration overall since the threads can fully utilize each available core.

b) [2 points] To support the highest load in the trace (125K QPS) without violating the 1ms latency SLO, how many memcached threads (T) and CPU cores (C) will you need?

Answer:
From our plot it is clear that the only configuration that can fulfill the SLO is $T = 2$ threads and $C = 2$ cores.

c) [1 point] Assume you can change the number of cores allocated to memcached dynamically as the QPS varies from 5K to 125K, but the number of threads is fixed when you launch the memcached job. How many memcached threads (T) do you propose to use to guarantee the 1ms 95th percentile latency SLO while the load varies between 5K to 125K QPS?

Answer:
From the plot we can see that when using $T = 1$, it is impossible to guarantee the 1ms 95th percentile latency SLO when the load is greater than 70K QPS. On the other hand, when using $T = 2$, we can guarantee the SLO if we use the following policy. If the load is less than 32K QPS we use a single core and if it's greater than 32K QPS we use two.

d) [8 points] Run memcached with the number of threads T that you proposed in (c) and measure performance with $C = 1$ and $C = 2$. Use the aforementioned `mcperf` command to sweep QPS from 5K to 125K.

Measure the CPU utilization on the memcached server at each 5-second load time step. Plot the performance of memcached using 1-core ($C = 1$) and using 2 cores ($C = 2$) in **two separate graphs**, for $C = 1$ and $C = 2$, respectively. In each graph, plot achieved QPS on the x-axis, ranging from 0 to 130K. In each graph, use two y-axes. Plot the 95th percentile latency on the left y-axis. Draw a dotted horizontal line at the 1ms latency SLO. Plot the CPU utilization (ranging from 0% to 100% for $C = 1$ or 200% for $C = 2$) on the right y-axis. For simplicity, we do not require error bars for these plots.

Plots:

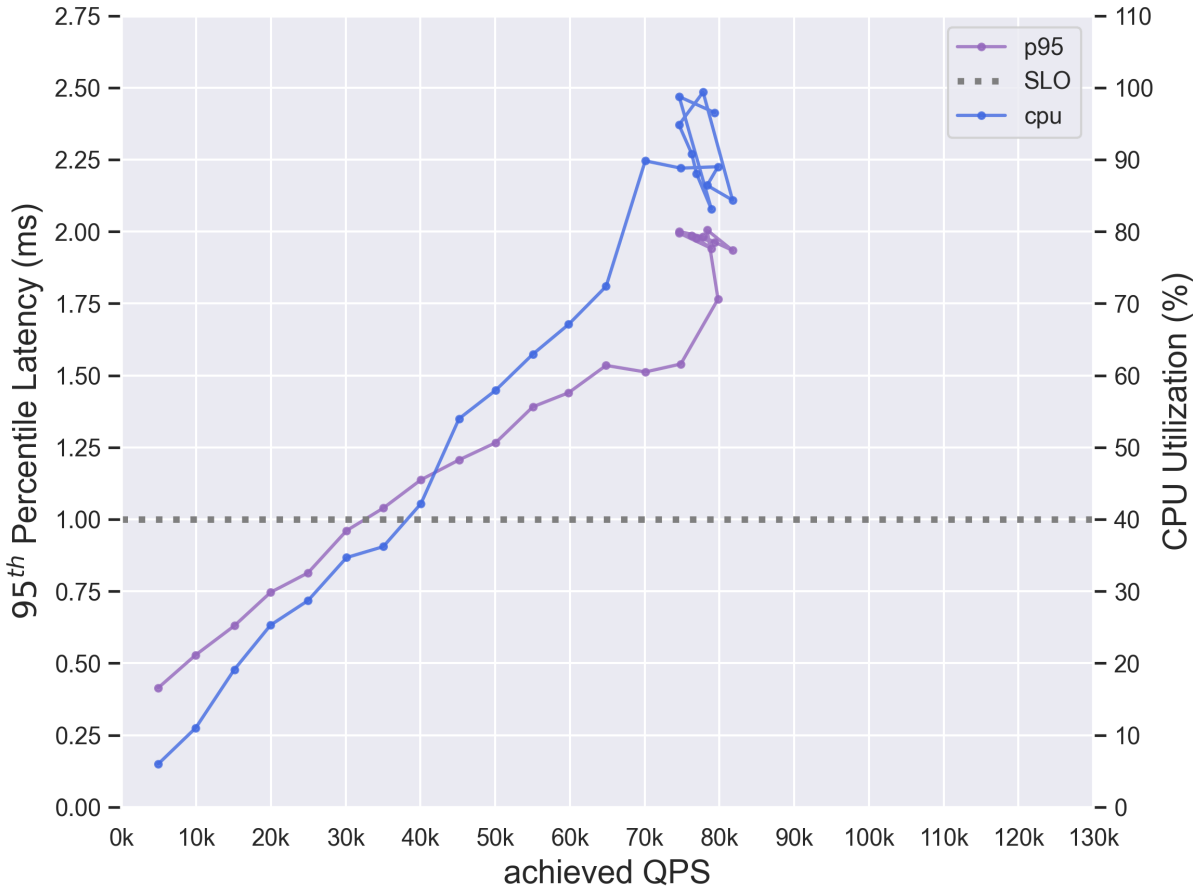

Figure 5: `memcached` latency and CPU utilization for increasing achieved QPS values, for $T = 2$ and $C = 1$.

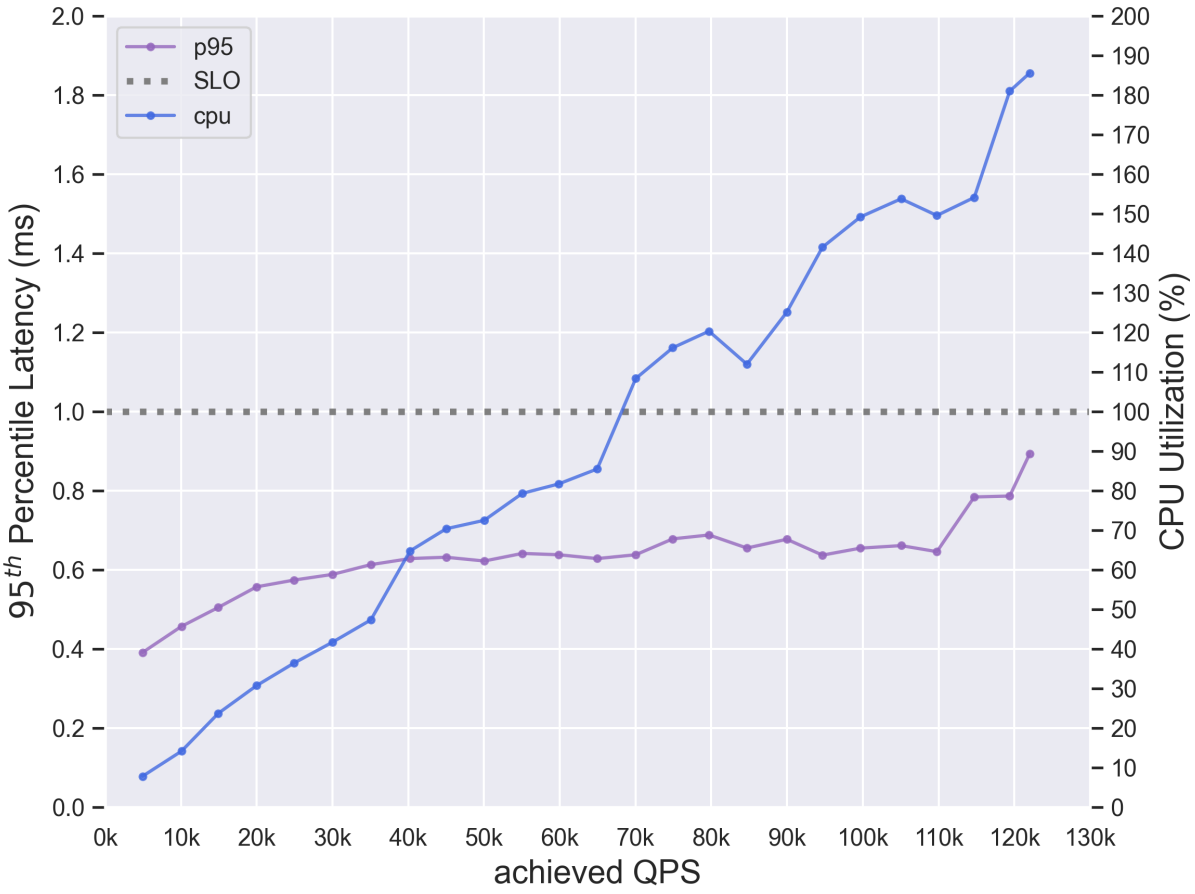

Figure 6: `memcached` latency and CPU utilization for increasing achieved QPS values, for $T = 2$ and $C = 2$.

2. **[17 points]** You are now given a dynamic load trace for memcached, which varies QPS randomly between 5K and 100K in 10 second time intervals. Use the following command to run this trace:

```
$ ./mcperf -s INTERNAL_MEMCACHED_IP --loadonly
$ ./mcperf -s INTERNAL_MEMCACHED_IP -a INTERNAL_AGENT_IP \
           --noload -T 16 -C 4 -D 4 -Q 1000 -c 4 -t 1800 \
           --qps_interval 10 --qps_min 5000 --qps_max 100000
```

   Note that you can also specify a random seed in this command using the `--qps_seed` flag.

   For this and the next questions, feel free to reduce the mcperf measurement duration (`-t` parameter, now fixed to 30 minutes) as long as you have at the end at least 1 minute of memcached running alone.

   Design and implement a controller to schedule memcached and the benchmarks (batch jobs) on the 4-core VM. The goal of your scheduling policy is to successfully complete all batch jobs as soon as possible without violating the 1ms 95th percentile latency for memcached.

Your controller should not assume prior knowledge of the dynamic load trace. You should design your policy to work well regardless of the random seed. The batch jobs need to use the native dataset, i.e., provide the option `-i native` when running them. Also make sure to check that all the batch jobs complete successfully and do not crash. Note that batch jobs may fail if given insufficient resources.

Describe how you designed and implemented your scheduling policy. Include the source code of your controller in the zip file you submit.

- Brief overview of the scheduling policy (max 10 lines):

  Answer:
  `memcached` server needs 2 cores (cores 0 and 1) to handle large loads without violating the SLO. Our scheduler computes the `memcached` CPU utilization, and if it drops below a certain threshold we reduce `memcached` cores to 1 (core 0 only), allocating core 1 to a parsec job. As soon as `memcached` CPU utilization goes over a different threshold, to avoid violating the SLO, we have to give core 1 back to the `memcached` server. Our scheduler runs the jobs dedup, vips and radix sequentially, on core 1, whenever it is not requested by the `memcached` server, pausing and resuming the respective container when needed. Meanwhile, the rest of the jobs are run in sequence using cores 2 and 3. As soon as either list of jobs finish, the remaining jobs will use cores 2, 3 and potentially core 1 when it's not requested by the `memcached` server.

- How do you decide how many cores to dynamically assign to memcached? Why?

  Answer:
  While `memcached` is running we can change the number of cores but cannot change the number of threads without restarting. So utilizing the knowledge from the previous questions we know that having two threads $T = 2$ is necessary to handle large workloads. In Figure 5, we can see that the SLO is violated for $\approx$ 30k QPS (C=1 T=2). However, CPU utilization is different for 30k QPS for 1 vs 2 cores. For this reason, our scheduler measures `mc_server` CPU utilization every 250 ms and if that exceeds or drops below a respective threshold, we change the cores accordingly. If `mc_server` uses 2 cores, the condition to reduce to 1 core is `cpu_util` $< 45\%$. If `mc_server` uses 1 core, the condition to increase to 2 cores is `cpu_util` $\geq 34.68\%$.

- How do you decide how many cores to assign each batch job? Why?

  Answer:
  Below is the initial and default assignment of cores for each job. The issue we built our policy around from was how to best utilize the 2nd core of `mc_server` as it won't always be available. If the randomness in the load is almost uniform from $5k - 100k$ QPS (mean load: $\approx 52.5k$), we expect that approx 30% of the total run time the first core will be available (soft bound, i.e. the time where QPS $\leq$ 30k) and thus assigned the jobs that take less time in general to that core, allocating the other 2 cores for the larger jobs. Finally, as previously explained, whenever either group of jobs finishes the other group will utilize the extra cores and instead of pausing containers running on the 2nd core of memcached, we then simply limit the cores of the respective jobs to 2 and 3, using the core 1 only when available[4].

[4] Usually jobs running on cores 2 and 3 finish first, but on a few runs the first group happened to finish first.

- blackscholes: 2 cores. This is a long job whose performance is CPU bound [1] and we need it finished as fast as possible.
- canneal: 2 cores. This is a long job who does not benefit a lot from multiple cores but is sensitive to LLC SoI and thus needs the 2 cores to finish as soon as possible.
- dedup: 1 core. This is a short job that benefits little from multithreading and thus can finish early when running on the shared core of `mc_server`.
- ferret: 2 cores. This is a long job who benefits significantly from using 2 threads but is also sensitive to CPU SoI and highly sensitive to LLC SoI, and thus needs the 2 cores to finish as soon as possible.
- freqmine: 2 cores. This is a long job who benefits significantly from using 2 threads but is also sensitive to CPU SoI and thus needs the 2 cores to finish as soon as possible.
- radix: 1 core. This is a short job that is not sensitive to most SoIs and thus can effectively run on the shared core of `mc_server`. Note that this is the last job to run on that core as it benefits significantly from multithreading, such that if the other jobs running on cores 2 and 3 finish, it can utilize that core and achieve better performance.
- vips: 1 core. This is a short job and does not benefit much from 2 cores, thus making a good fit for the shared core of `mc_server`.

- How many threads do you use for each of the batch job? Why?

  Answer: Repeating our comments from Part 3:
  In general, it is a good idea to have as many threads as the number of available cores [1], or close to that number (always `num_cores` $\leq$ `num_threads`), to allow most sytems to fully utilize multithreaded parallelization, while not adding too much overhead.
  - blackscholes: 2 threads. Our reasoning here is the same as for Part 3. `blackscholes` does not achieve significant speedup from multithreading, but we know that its performance is limited by the number of flops the CPU may perform [1].
  - canneal: 4 threads. This job is running on 2 cores, having a much larger number of threads would not be advisable since it wouldn't allow our system to properly utilize parallelization. We use 4 threads instead of 2 because, from our previous analysis, we know that `canneal` benefits significantly from multithreading, but is also not sensitive to CPU SoI and can effectively share the available cores across the threads.
  - dedup: 1 thread. From previous analysis, we know that `dedup` doesn't significantly benefit from parallelization so we choose to run it with the same amount of threads as the amount of cores it is running on as it is moderately sensitive to CPU SoI.
  - ferret: 2 threads. From previous analysis, we know that `ferret` is highly sensitive to CPU SoI so we choose to run it with the same amount of threads as the amount of cores it is running on as it needs to fully utilize its cores.
  - freqmine: 2 threads. From previous analysis, we know that `freqmine` is highly sensitive to CPU SoI so we choose to run it with the same amount of threads as the amount of cores it is running on as it needs to fully utilize its cores.
  - radix: 4 threads. From our previous analysis `radix` gives us a significant speedup and benefits from multithreading. However, it mostly runs on 1 core, but we chose to run it last in that group of jobs such that if the rest of the cores become available,

it will be able to fully utilize them. Also, the larger number of threads is not a problem as the job is not sensitive to CPU SoI, and thus can effectively share the CPU core(s) across threads.

- vips: 1 thread. From previous analysis, we know that `vips` is moderately sensitive to CPU SoI and thus would be a good idea not to use more threads than cores, so it will be able to fully utilize the CPU and finish as early as possible.

- Which jobs run concurrently / are collocated and on which cores? Why?
  Answer:

  - In core zero we always run `memcached` alone with no other jobs.
  - In core one, when `memcached` is not running we run dedup, vips and radix sequentially in that order. We don't run anything concurrently with `memcached` in order to not interfere with the server. If CPU utilization is tampered with by the other jobs we will not be able to use it as a guide of when to switch between using 2 cores or 1 core for `memcached`. These are the shortest jobs and as we expect that core to be available approximately less than 30% of time, the jobs can finish approximately (or usually shortly after) the longer jobs running on the other 2 cores.
  - In cores two and three, we run blackscholes, canneal, ferret, and freqmine sequentially in that order. They are run sequentially as they are the longest jobs in our working set and can benefit a lot from fully using the 2 available cores.

- In which order did you run the batch jobs? Why?
  Order (sorted): blackscholes, dedup, canneal vips, ferret, freqmine, radix

  Why:
  As explained above, the order we chose to execute our jobs is split them first into 2 groups, based on their total run time (long $>$ 1.5 minutes, short $<$ 1 minute) as the 1 core will be available for parsec jobs less than 30% of the total run time. The order of the jobs of each group is sorted based on their multithreading benefit in increasing order, such that if the other cores become available for that job, it will be able to utilize them. The detailed order is therefore as follows:

  - Group A (running on core 1 when available): dedup, vips, radix
  - Group B (running on cores 2 and 3): blackscholes, canneal, ferret, freqmine.

  As radix is the short job and freqmine is the long job that benefit the most from multithreading respectively.

- How does your policy differ from the policy in Part 3? Why?
  Answer:
  Firstly, comparing the resources - VMs, that were available to us for both parts, in Part 3 we are have significantly more resources, since we have 3 VMs instead of 1 as we do in Part 4. Having more VMs gave us the flexibility to run `memcached` in a seperate node than most of the other jobs and thus prevent SLO violations without dynamic scheduling.
  In Part 4, we have a single VM so we have to share the available resources. Thus, to avoid SLO violations, we have at less than 30% of the total run time 3 available CPU cores and 2 cores otherwise. Since the node is of type `n2d-highmem-4`, we did not consider any memory limits and problems.

The dynamic policy with the limited amount of resources focuses on running the long jobs on as many resources as possible while dynamically adjusting the cores for `mc_server`. Our previous static policy focused on providing the long jobs the most suitable resources and run in parallel on the same node the jobs that had reduced pairwise interference, while the `mc_server` would run on 1 specific core always.

- How did you implement your policy? e.g., docker cpu-set updates, taskset updates for memcached, pausing/unpausing containers, etc.

  Answer:
  We use all of the aforementioned methods, building a controller with a similar interface to the scheduling logger provided, in Python, and using Docker SDK. Specifically:
  - **docker run and remove**: We use Docker python SDK to run Docker containers and remove them as soon as they exit. Each starting container is detached and we check their status frequently to see if the execution is on-going or finished, before removing the container with `container.remove()`.
  - **docker update**: We use docker update to update the cores of a job, if the 1 group of jobs finishes before the other to fully utilize all available cores when available.
  - **docker pause and unpause**: we use this functionality to pause the containers for the jobs that are running on the shared core of `mc_server` when the server's CPU utilization exceeds the aforementioned threshold. When the CPU utilization falls again we unpause the containers and resume the jobs in the core that `memcached` is no longer running on.
  - **psutil**: we use this library to retrieve information about the CPU utilization per core, as well as the CPU utilization of the `mc_server` using:
    `psutil.cpu_percent(interval=None, percpu=True)`
    In both cases we used `interval=None` to compare system CPU times elapsed since last call or module import, returning immediately (non blocking).
  - **taskset updates for memcached**: we use `taskset` to assign CPU cores to `memcached` based on our monitoring of its CPU utilization. We run this command through our scheduler using `subprocess` library:
    `subprocess.run(f"sudo taskset -a -cp {cpus} {memcached_pid}".split(" "))`

- Describe the design choices, ideas and trade-offs you took into account while creating your scheduler (if not already mentioned above):

  Answer:
  Our ideas are thoroughly explained above. The tradeoffs we took into account mostly concerned the CPU utilization threshold of the server to switch from and to using 2 cores. Increasing the threshold to switch from 1 to 2 cores can result in less runtime for the jobs but more potential SLO violations. Another tradeoff was the estimated availability of that core: in practice, it usually less than 30%. In the absolute worst case of 0% availability, we would expect increased delay in the total runtime of our jobs up to the sum of the time needed for the short jobs group to sequentially run on 2 cores. Finally, we assumed that the resources needed for our scheduler to be insignificant and did not try to limit the scheduler in specific resources. To reduce network overhead, we prepulled and cached all container images so they can start almost immediately.
  To improve our scheduler, we could try to first design a more complicated policy sharing sub-cpu cores across jobs. Also, we could try to find harder threshold bounds for the server's 2nd core allocation. Finally, we can run the job status checking in a separate thread (i.e. like a heartbeat), to start containers even sooner.

3. **[23 points]** Run the following `mcperf` memcached dynamic load trace:

```
$ ./mcperf -s INTERNAL_MEMCACHED_IP --loadonly
$ ./mcperf -s INTERNAL_MEMCACHED_IP -a INTERNAL_AGENT_IP \
           --noload -T 16 -C 4 -D 4 -Q 1000 -c 4 -t 1800 \
           --qps_interval 10 --qps_min 5000 --qps_max 100000 \
           --qps_seed 3274
```

Measure memcached and batch job performance when using your scheduling policy to launch workloads and dynamically adjust container resource allocations. Run this workflow 3 separate times. For each run, measure the execution time of each batch job, as well as the latency outputs of memcached. For each batch application, compute the mean and standard deviation of the execution time [5] across three runs. Compute the mean and standard deviation of the total time to complete all jobs - the makespan of all jobs. Fill in the table below. Also, compute the SLO violation ratio for memcached for each of the three runs; the number of data points with 95th percentile latency $>$ 1ms, as a fraction of the total number of datapoints. The SLO violation ratio should be calculated during the time from when the first batch-job-container starts running to when the last batch-job-container stops running.

job name	mean time [s]	std [s]
blackscholes	63.67	2.89
canneal	151.67	2.52
dedup	29.33	0.58
ferret	202	6.56
freqmine	258.33	2.52
radix	23	0.0
vips	103.33	2.52
total time	706.33	14.57

Answer:
Our SLO violations are as follows:

run	number of violations / number of datapoints	percentage
run 1	0/90	0.0%
run 2	1/90	1.11%
run 3	1/90	1.11%

Notice that the SLO violation occurs while the job running on the shared core of `mc_server` starts and stops many times within a small interval, meaning that the indicated CPU utilization is between the 2 thresholds at the time and does not stabilize even with our stabilization mechanism, resulting in an overhead for the server.

Include six plots – two plots for each of the three runs – with the following information. Label the plots as 1A, 1B, 2A, 2B, 3A, and 3B where the number indicates the run and the letter indicates the type of plot (A or B), which we describe below. In all plots, time will be on the x-axis and you should annotate the x-axis to indicate which benchmark (batch) application

[5] Here, you should only consider the runtime, excluding time spans during which the container is paused.

starts executing at which time. If you pause/unpause any workloads as part of your policy, you should also indicate the timestamps at which jobs are paused and unpaused. All the plots will have have two y-axes. The right y-axis will be QPS. For Plots A, the left y-axis will be the 95th percentile latency. For Plots B, the left y-axis will be the number of CPU cores that your controller allocates to memcached. For the plot, use the colors proposed in this template (you can find them in `main.tex`).

Plots:

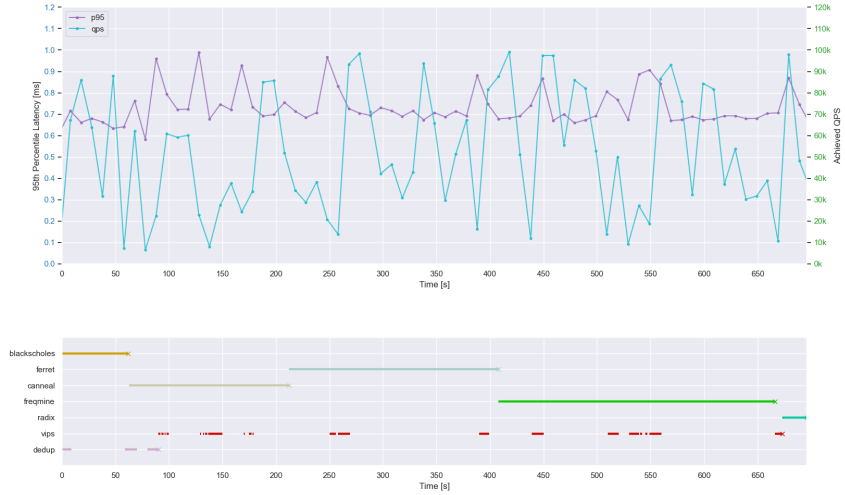

Figure 7: 1A.

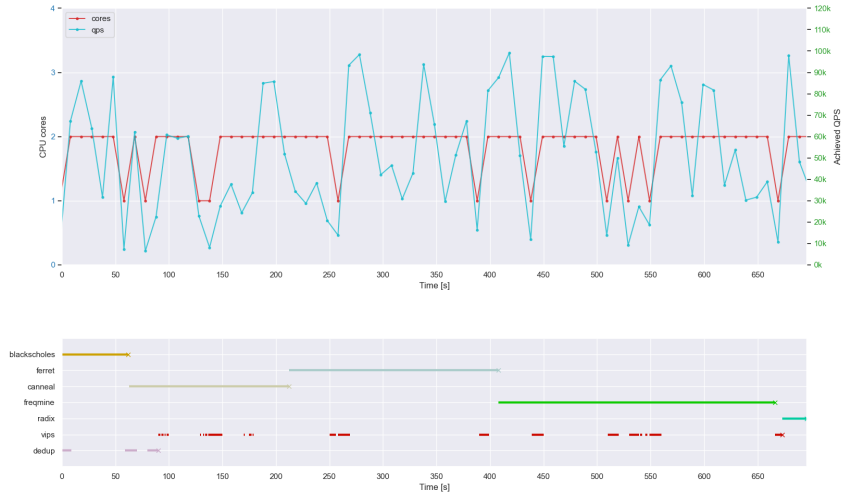

Figure 8: 1B.

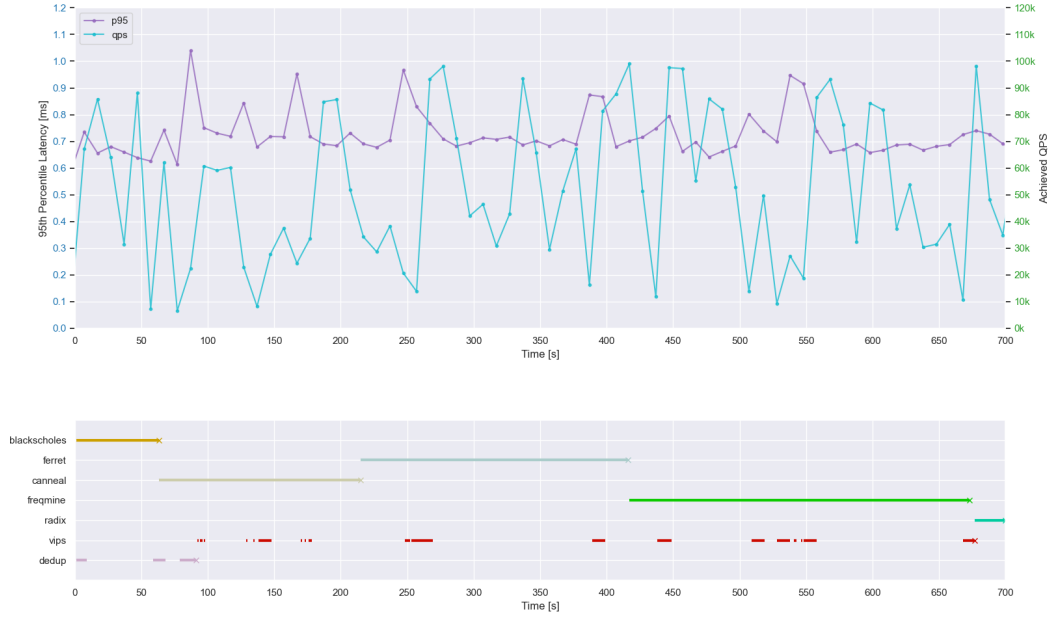

Figure 9: 2A.

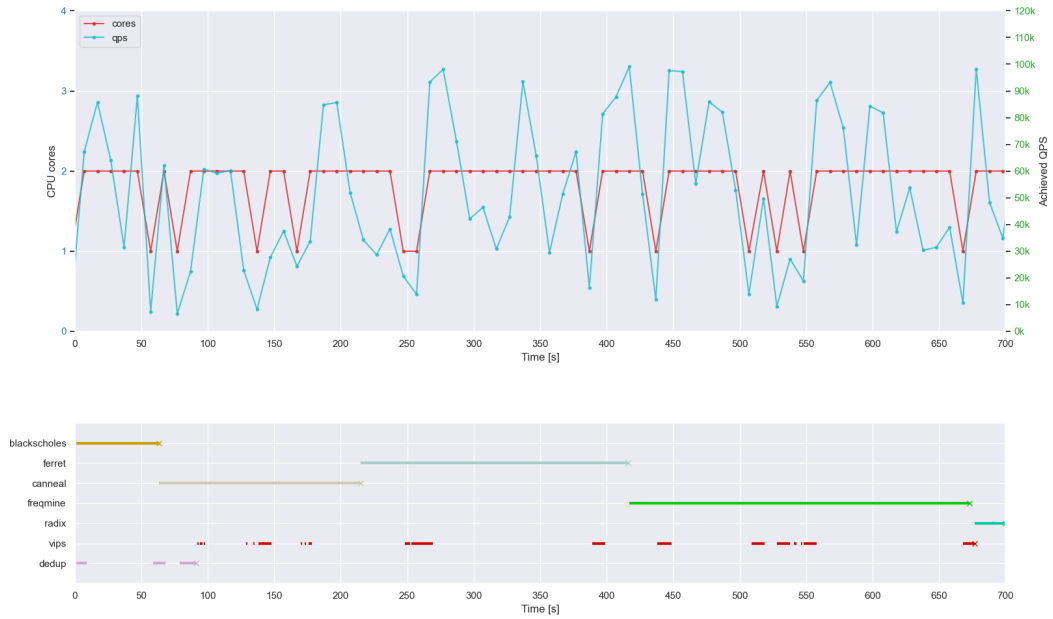

Figure 10: 2B.

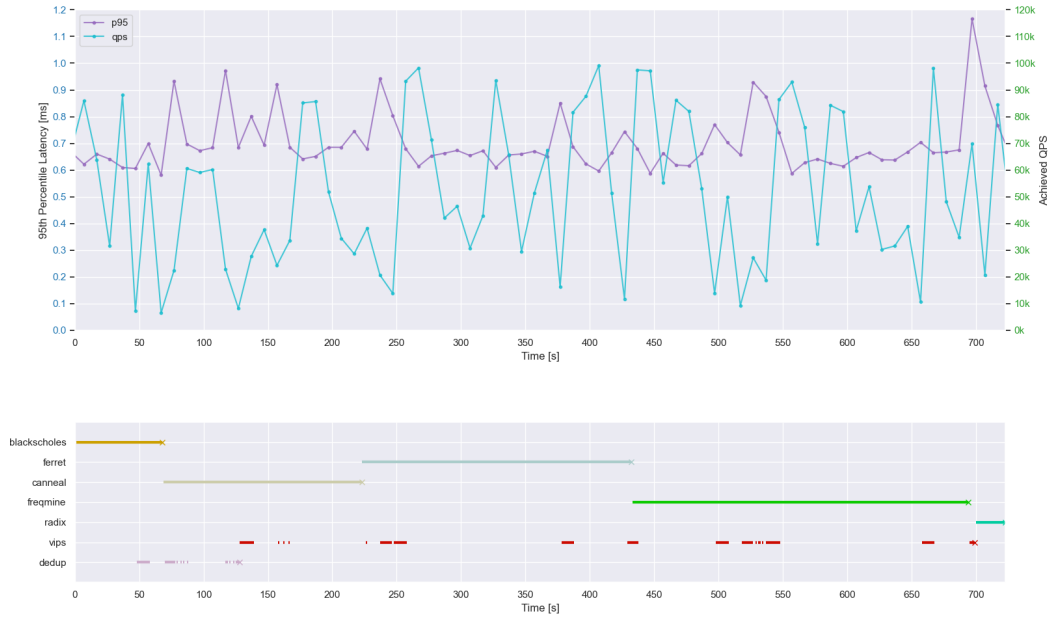

Figure 11: 3A.

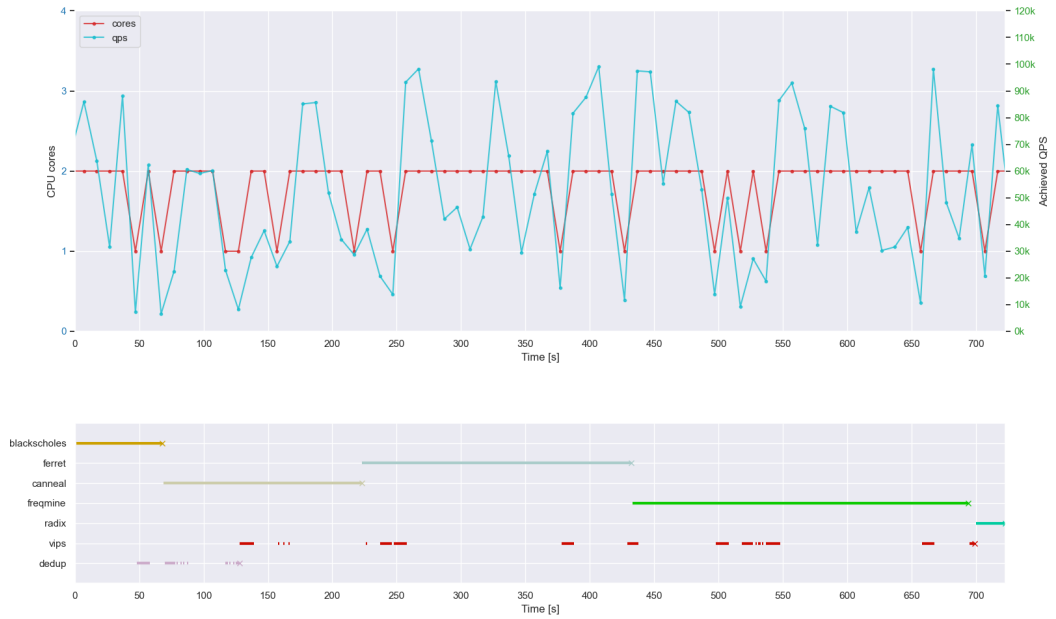

Figure 12: 3B.

4. **[16 points]** Repeat Part 4 Question 3 with a modified `mcperf` dynamic load trace with a 5 second time interval (`qps_interval`) instead of 10 second time interval. Use the following command:

```
$ ./mcperf -s INTERNAL_MEMCACHED_IP --loadonly
$ ./mcperf -s INTERNAL_MEMCACHED_IP -a INTERNAL_AGENT_IP \
           --noload -T 16 -C 4 -D 4 -Q 1000 -c 4 -t 1800 \
           --qps_interval 5 --qps_min 5000 --qps_max 100000 \
           --qps_seed 3274
```

You do not need to include the plots or table from Question 3 for the 5-second interval. Instead, summarize in 2-3 sentences how your policy performs with the smaller time interval (i.e., higher load variability) compared to the original load trace in Question 3.

Summary:
When using a 5-second interval, the total execution time of the jobs is similar to before. We notice that jobs running on core 1 are paused and unpaused more frequently and that we have increased SLO violation ration by $\approx \times 3.5$ (1.11% to 3.89%).

What is the SLO violation ratio for memcached (i.e., the number of datapoints with 95th percentile latency $>$ 1ms, as a fraction of the total number of datapoints) with the 5-second time interval trace? The SLO violation ratio should be calculated during the time from when the first batch-job-container starts running to when the last batch-job-container stops running.

Answer:
When using a 5-second time interval trace, we measure an SLO violation ratio of $\frac{7}{180} = 3.89\%$ for the duration the containers are active.

What is the smallest `qps_interval` you can use in the load trace that allows your controller to respond fast enough to keep the memcached SLO violation ratio under 3%?

Answer:
The smallest integer possible `qps_interval` is 6 seconds, since it results in a SLO violation ratio of $\frac{4}{150} = 2.67\%$, while 5 seconds resulted in a violation ratio of 3.89%.

What is the reasoning behind this specific value? Explain which features of your controller affect the smallest `qps_interval` interval you proposed.

Answer:
Using a smaller `qps_interval` window increases the variability of the `mc_server` load, resulting in more switches back-n-forth from 1 to 2 cores in total for the server cores, which is the phenomenon we noticed in Figures 8, 10, 12 that causes the SLO violations. Therefore, the `qps_interval` affects our scheduler's feature that decides when to allocate 2 cores for the server and when to allocate the 2nd core for the parsec jobs.

To choose the value of 6, we noticed that a value of 5 was already surpassing the 3% SLO violation ratio limit. We therefore chose decrease the load variability by increasing the `qps_interval` to 6 seconds, and found that with this value the SLO violation ratio remains within the desired bound. Since `qps_interval` is type of DOUBLE, to get an even smaller value we could try with floats (i.e. 5.5) but for the sake of our resources we didn't attempt to further push it.

Use this `qps_interval` in the command above and collect results for three runs. Include the same types of plots (1A, 1B, 2A, 2B, 3A, 3B) and table as in Question 3.

job name	mean time [s]	std [s]
blackscholes	66.67	2.52
canneal	154	7.00
dedup	29	1.73
ferret	203.33	8.08
freqmine	250.67	2.93
radix	24.67	1.53
vips	104.33	3.21
total time	695.67	15.37

Plots:
The plots follow the same structure as before. Notice however with this smaller `qps_interval` that the QPS load variation has increased. Also, the variance in the total runtime has increased across the 3 runs. For each of the 3 runs, we only had 4 SLO violations in 150 total datapoints. The mean total time has reduced by 11 seconds compared to `qps_interval` = 10,

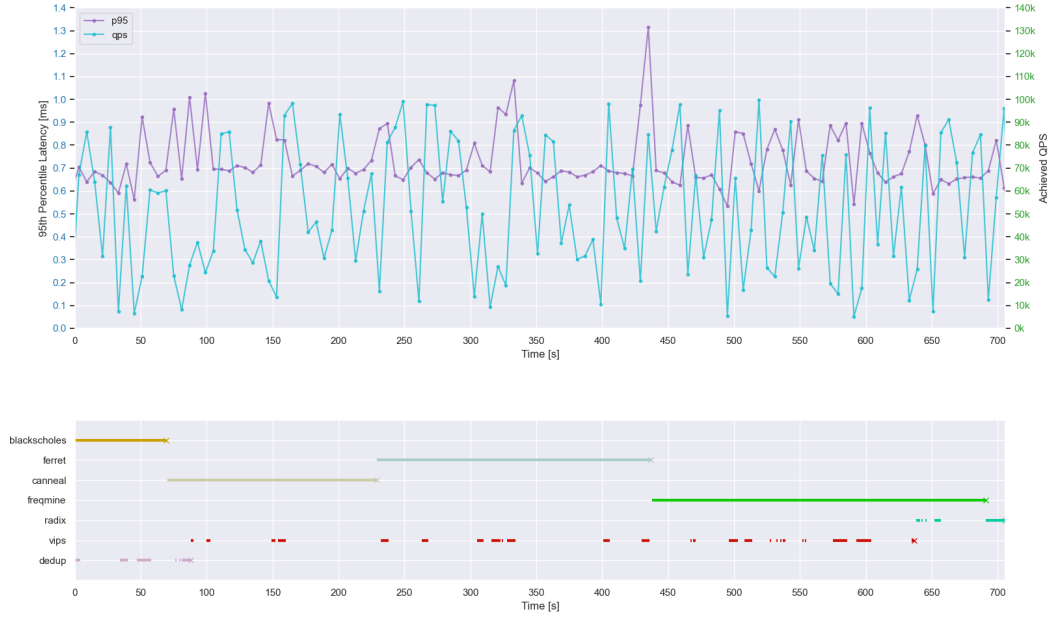

Figure 13: 1A.

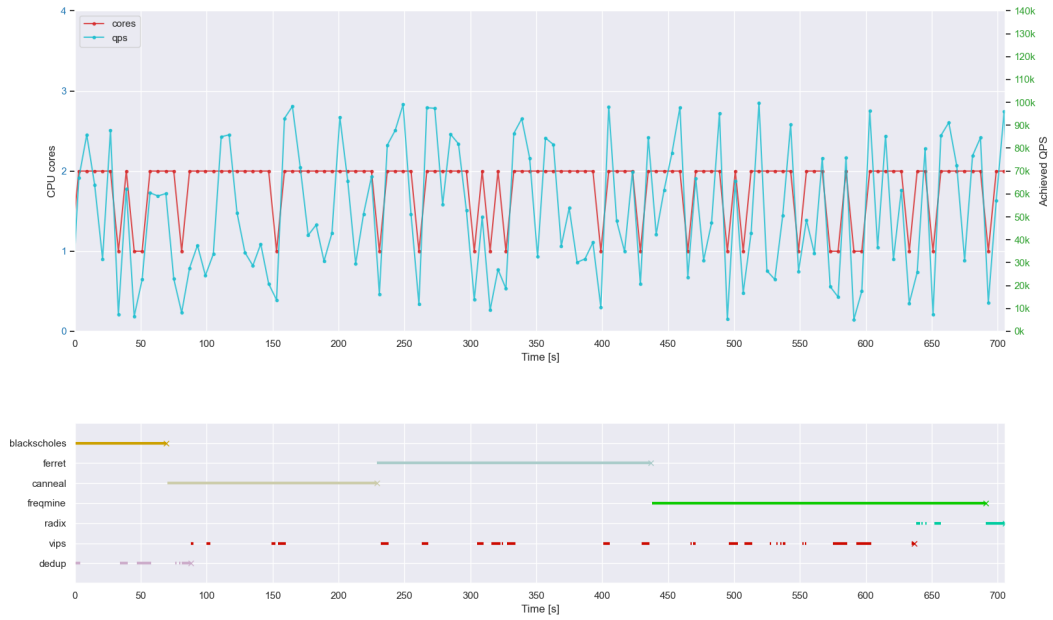

Figure 14: 1B.

Figure 15: 2A.

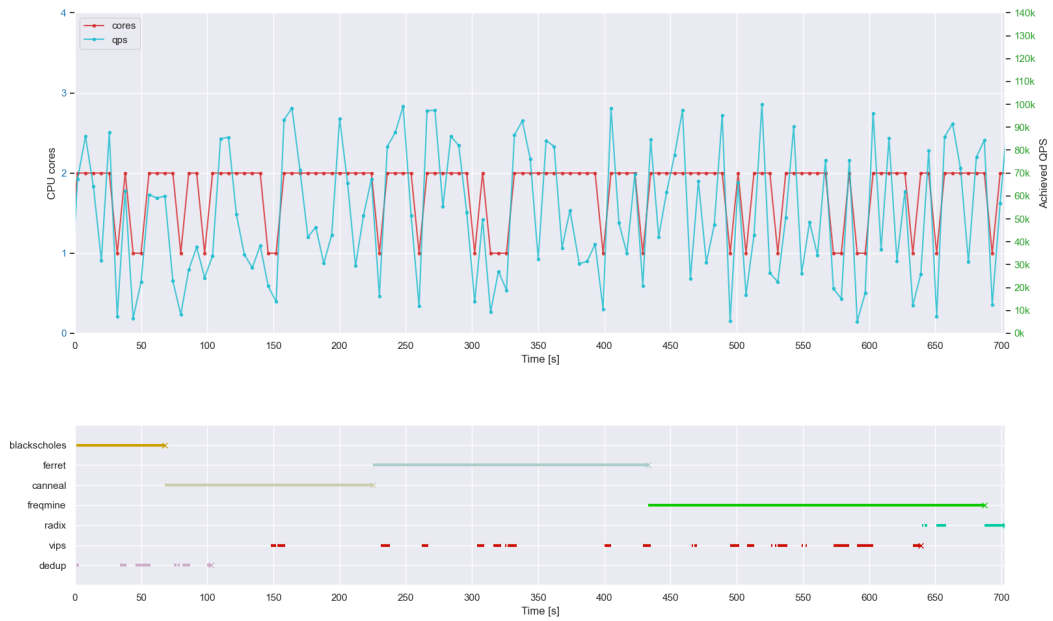

Figure 16: 2B.

Figure 17: 3A.

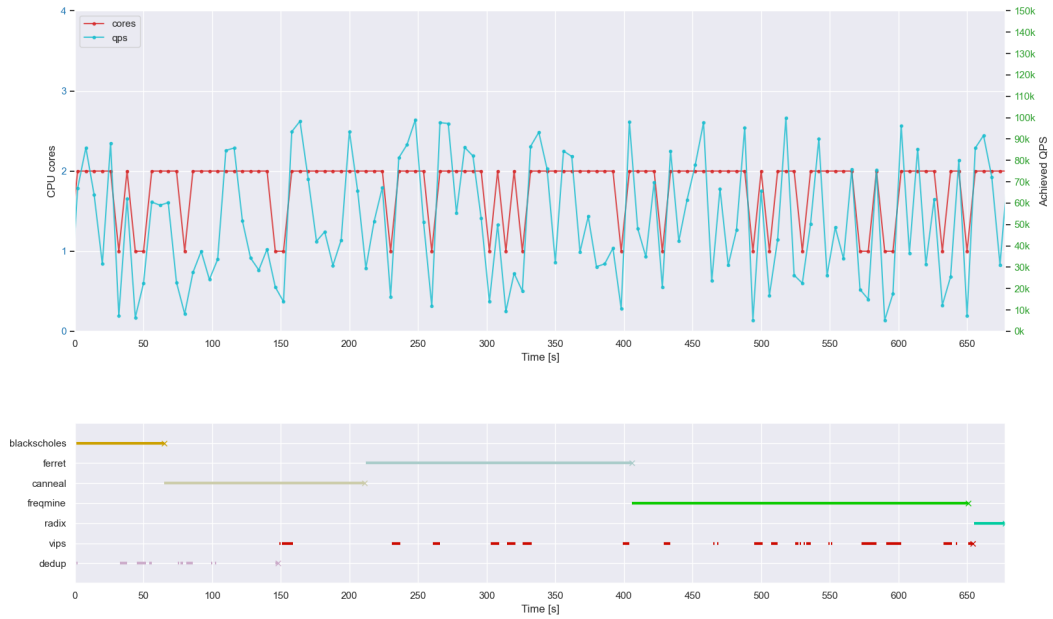

Figure 18: 3B.

References

[1] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, page 72–81, New York, NY, USA, 2008. Association for Computing Machinery.

[2] ETH-EASL. Memcached Dynamic Github Repository. https://github.com/eth-easl/memcache-perf-dynamic.