Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Cloud computing architecture

Semester project report

Group 5
Berke Egeli – 20-948-220
Floris Westermann – 16-944-662
Julia Bazinska – 20-942-843

Systems Group
Department of Computer Science
ETH Zurich
June 16, 2021

Instructions

Please do not modify the template, except for putting your solutions, names and legi-NR.

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 PARSEC job, as well as the latency outputs of memcached running with a steady client load of 30K QPS. For each PARSEC application, compute the mean and standard deviation of the execution time across three runs. Also compute the mean and standard deviation of the total time to complete all jobs. Fill in the table below. Finally, compute the SLO violation ratio for memcached for the three runs; the number of datapoints with 95th percentile latency ¿ 2ms, as a fraction of the total number of datapoints. Do three plots (one for each run) of memcached p95 latency (y-axis) over time (x-axis) with annotations showing when each parsec job started.

job name	mean time [s]	std [s]
dedup	20.67	0.47
blackscholes	165.67	3.86
ferret	250.00	2.16
freqmine	193.00	1.41
canneal	270.33	6.13
fft	121.33	6.13
total time	372.67	14.43

The requested plots can be found in Figure 1. Violation rate for all three runs was 0.

2. **[17 points]** Describe and justify the "optimal" scheduling policy you have designed. This is an open question, but you should at minimum answer the following questions:

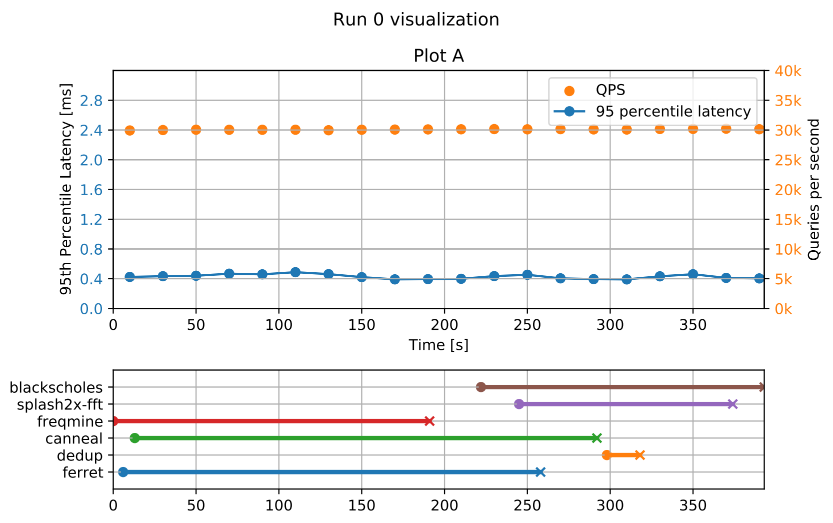
   - Which node does memcached run on?
   - Which node does each of the 6 PARSEC apps run on?
   - Which jobs run concurrently / are collocated?
   - In which order did you run 6 PARSEC apps?
   - How many threads you used for each of the 6 PARSEC apps?

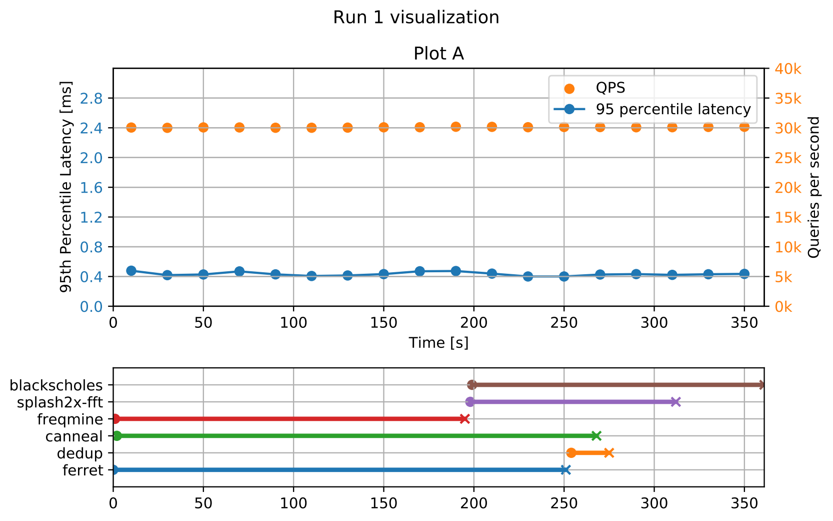
   Describe how you implemented your scheduling policy. Which files did you modify or add and in what way? Which Kubernetes features did you use? Please attach your modified/added YAML files, run scripts and report as a zip file. **Important: The search space of all the possible policies is exponential and you do not have enough credit 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.**

As seen in the previous parts, memcached is not safe to collocate with any PARSEC job, therefore, we run memcached by itself on the 2 core VM node-a-2core.

Ferret, canneal, and dedup run on the 8 core VM node-c-8core. Freqmine, blackscholes, and splash2x-fft run on the 4 core VM node-b-4core.

Canneal runs concurrently with ferret, dedup, freqmine, blackscholes, and splash2x-fft. It is collocated with both ferret and dedup. Ferret runs concurrently with canneal, freqmine, blackscholes





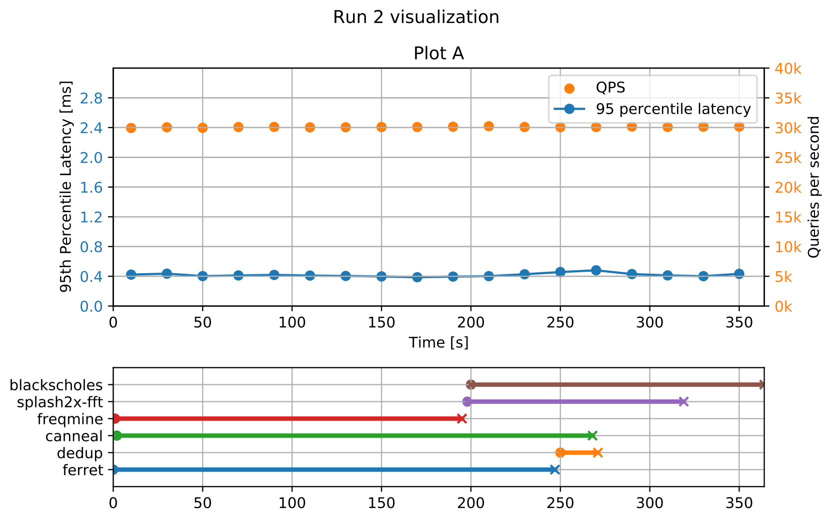


Figure 1: Static schedule visualization.

and splash2x-fft. It is collocated with canneal. Dedup runs concurrently with canneal, blackscholes and splash2x-fft. It is collocated with canneal. Freqmine runs concurrently with ferret and canneal. It is not collocated with other jobs. Blackscholes runs concurrently with splash2x-fft, ferret, canneal and dedup. It is collocated with splash2x-fft. Splash2x-fft runs concurrently with blackscholes, ferret, canneal and dedup. It is collocated with blackscholes.

The static scheduler first runs ferret and canneal on 8 core VM. At the same time we also start running freqmine on the 4 core VM. Ferret ends before canneal does, so, in the freed up cores we start running dedup. Dedup and canneal end around the same time at the 5 minute mark. Freqmine ends around the 3 to 3.5 minutes mark. After it finishes we start running blackscholes and splash2x-fft together. Splash2x-fft runs for about 2 minutes and blackscholes runs for about 2.5 minutes.

We run ferret with 4, canneal with 2, dedup with 4, freqmine with 4, blackscholes with 2 and splash2x-fft with 2 threads.

In total we have modified 7 YAML files, 6 of them are used by the PARSEC jobs and the last one is used by memcached. In PARSEC YAML files we have modified the following lines: spec.template.spec.containers.args and spec.template.spec.nodeSelector.cca-project-nodetype. The latter is modified to indicate which node a pod should be located at. We used Kubernetes' grouping mechanisms to achieve this, node selector in particular since label was already defined in the YAML file for part 3. The former line is modified to specify the CPU affinity of PARSEC jobs and how many threads we want to run them with. CPU affinity of ferret is 0-5, canneal is 6-7, dedup is 0-3, freqmine is 0-3, blackscholes is 0-1 and splash2x-fft is 2-3. We assign more CPUs to ferret than the number of threads we specify in its YAML file. This is because ferret is a CPU intensive job that spawns additional threads than the 4 we specified and it slows down considerably if we assign it only 4 CPUs. Additionally in canneal, dedup, ferret and freqmine we set resource requests and limits for CPU and for splash2x-fft we set memory request and limits. However, these were experimental and putting the limits we chose did not affect execution speed. This is most probably the case because none of the jobs share any of the CPUs to begin with due to different CPU affinities. In memcached's YAML file, we modified the following: spec.containers.args and spec.nodeSelector.cca-project-nodetype. Memcached is modified to use the entire 2 core VM with 2 threads since it is not collocated with other jobs.

The primary idea of our static scheduling policy was to collocate jobs in a way that would minimize the interference with the jobs on the critical path. The critical path consists of freqmine and blackscholes. Ferret, canneal, and splash2x-fft were around 20% slower. Dedup was around 50% slower but it was not in the critical path. Blackscholes was around 7% slower. Freqmine was around 50-60% slower because it was executing in the 4 core VM. This does theoretically conflict with our initial goal of minimizing the interference with the critical path. However, we decided against running it on the 8 core VM because either we would have to postpone its execution until after ferret stopped executing or we would have to collocate it with ferret, however, these two jobs have very similar resource usages and they are already relatively slow compared to other jobs. That is why we decided to run it alone in the 4 core VM. This caused our critical path to be freqmine and blackscholes and have a total execution time of around 6 minutes with no SLO violations.

Part 4 [76 points]

1. **[10 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. 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

For this question, use the following `mcperf` command to vary QPS from 5K to 120K:

```
$ ./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 5000:120000:5000
```

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.

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.

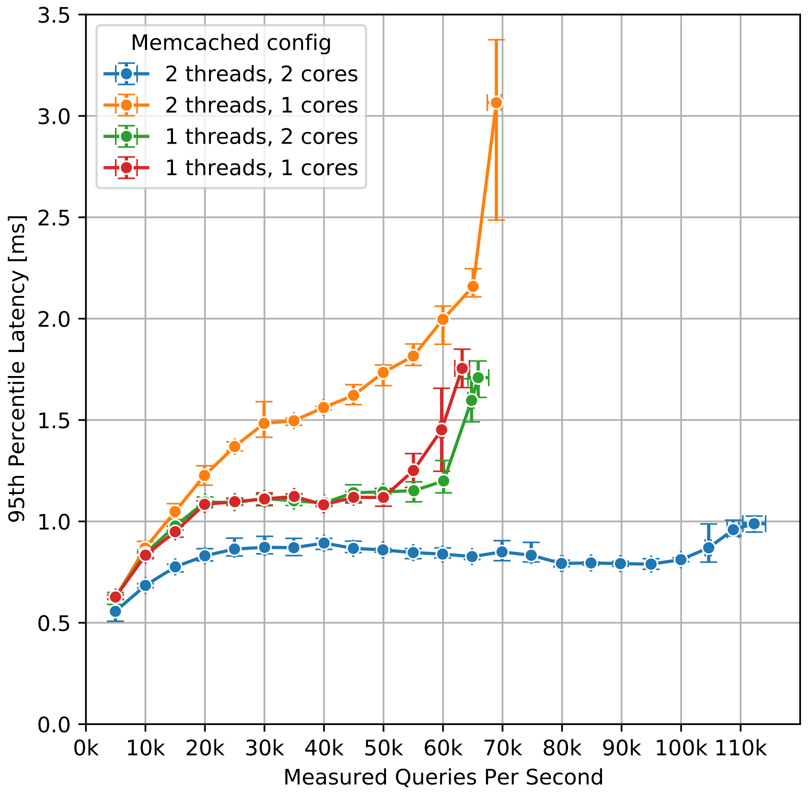


Figure 2: Memcached performance for Part 4 Question 1. Datapoints for which the measured QPS was more than 5000 lower than the target QPS were omitted for readability. Presented results are aggregated over 3 runs.

From Figure 2 we can conclude that, in order to have optimal performance, the number of threads should match the number of cores available. Running 1 thread on any number of cores is essentially the same thing because the thread will only be able to utilize one of the cores at a given time. The latency for running 2 threads on 1 core is the worst as in that case both threads will be competing with each other for CPU time. Running 2 threads on 2 cores is the best allocation in these 4 scenarios as we give the most amount of resources to threads that can actually utilize those resources.

2. [**8 points**] Now assume you need to support a memcached request load that ranges from 5K to 100K QPS while guaranteeing a 2ms 95th percentile latency SLO.

   a) To support the highest load in the trace (100K QPS) without violating the 2ms latency SLO, how many memcached threads (T) and CPU cores (C) will you need? i.e., what value of T and C would you select?

   T = 2 and C = 2 is the only available configuration that can support up to 100k, therefore, we would select those values to prevent SLO violations.

   b) Assume you can change the number of cores allocated to memcached dynamically as the QPS varies from 5K to 100K, 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 2ms 95th percentile latency SLO while the load varies between 5K to 100K QPS? i.e., what values of T would you select?

   T = 2 is the only viable option to prevent SLO violations. The memcached would be slower when it is running on a single core, however, we can easily increase the number of cores when the load increases and we get closer to violating the SLO.

   c) Run memcached with the number of threads T that you proposed in b) above and measure performance with $C = 1$ and $C = 2$. Use the following `mcperf` command to sweep QPS from 5K to 100K:

```
$ ./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 5000:100000:5000
```

   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 QPS on the x-axis, ranging from 5K to 100K. In each graph, use two y-axes. Plot the 95th percentile latency on the left y-axis. Draw a dotted horizontal line at the 2ms 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.

   Data is presented in Figure 3.

3. [**15 points**] You are now given a dynamic load trace for memcached, which varies QPS

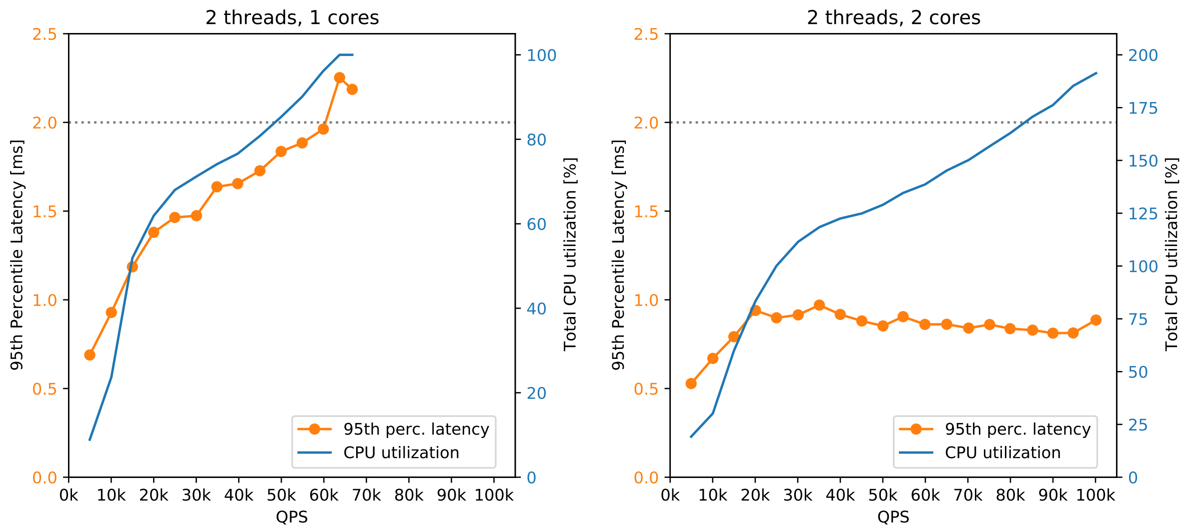


Figure 3: Performance of memcached for Question 2 running 2 threads on 1 core (left) and 2 threads on 2 cores (right). Datapoints for which the measured QPS was more than 5000 lower than the target QPS were omitted for readability.

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.

Design and implement a controller to schedule memcached and the PARSEC benchmarks on the 4-core VM. The goal of your scheduling policy is to successfully complete all PARSEC jobs as soon as possible without violating the 2ms 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.** Also make sure to check that all the PARSEC jobs complete successfully and do not crash. Note that PARSEC 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. To describe your scheduling policy, you should at minimum answer the following questions. For each, also **explain why**:

- How do you decide how many cores to dynamically assign to memcached?
- How do you decide how many cores to assign each PARSEC job?
- How many threads do you use for each of the PARSEC apps?
- Which jobs run concurrently / are collocated and on which cores?
- In which order did you run the PARSEC apps?

- How does your policy differ from the policy in Part 3?
- How did you implement your policy? e.g., docker cpu-set updates, taskset updates for memcached, pausing/unpausing containers, etc.

Memcached CPU cores. Our controller lets memcached use all cores of the VM. This was done to allow the underlying operating system scheduler to assign memcached as much cpu as it needs as quickly as possible. Memcached never appears to needs more than 2 cores to process all incoming requests.

PARSEC jobs resource affinity. Each PARSEC job is assigned a weight based on how many cores it maximally utilizes. This is known for each job and also has to be specified in the command to run the container.

A job can have weight 1, 2, or 3. Each weight class has a respective queue in which the jobs are kept. We schedule jobs by pausing and unpausing their containers. Each job is allowed to use any core it desires.

We schedule the containers based on the total load level of the VM and the load level of memcached. When less than a single core is used (i.e. total load $< 100\%$) the scheduler tries to run containers with a total weight of 4 (4 units). It will schedule 3 units if less than 2 cores are fully used, 2 units if less than 3 cores, and 1 unit if the VM is utilized less than 350%. The scheduler will make sure that it does not schedule more than 5 units at a time. This lets the VM run at maximal capacity.

We only pause containers when memcached starts showing more cpu utilization. When the total utilization exceeds 380% and memcached uses more than 60% we pause containers until only maximally 3 units remain. When memcached exceeds more than 150% CPU usage we pause everything but maximally 2 units.

Our implementation does not adjust the core affinity of the containers. This could prevent any cpu-contention for memcached. We did play with the idea and had prototypes that implement this. But in the end it seemed unnecessary: strictly assigning CPU-cores would have increased the overhead since we would also have to reassign cores depending on the utilization of each container. Instead we conveniently outsource this task to the operating system.

We distributed the task across the queues in the following way:

- queue 1: dedup, splash2x-fft
- queue 2: canneal, freqmine, blackscholes
- queue 3: ferret

Number of threads:

- dedup: 1 thread
- splash2x-fft: 1 thread
- blackscholes: 2 threads
- canneal: 2 threads
- freqmine: 2 threads
- ferret: 3 threads

Our policy only specifies an order between the start times of the containers since they are scheduled in the order of the queue. However, what containers run simultaneously or in what order they finish is not fixed since it depends on the load trace. The exact order of execution can be seen in Figures 4, 5, and 6.

Comparison to static scheduler. For the static scheduling task, we had more resources to perform the computation (three VMs vs just one). Since we have more limited resources in part 4, we could not just assign memcached separate resources to ensure the SLO is met. Thus, the scheduler manages the jobs in an entirely different way: memcached is assigned four cores (all available) that are shared with the other workloads. The jobs are released dynamically based on resource availability, not based on our predefined collocation strategy.

Implementation details.

For the implementation we used Python with Docker SDK for container management (starting, pausing, assigning CPUs). We used `taskset` to assigned CPU cores to memcached. For monitoring CPU utilization and memcached CPU utilization we used the psutil package for Python. Our prototypes have also used `renice` to give priority to memcached but that gave us no improvement gains.

4. [**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 42
```

Measure memcached and PARSEC 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 PARSEC job, as well as the latency outputs of memcached. For each PARSEC application, compute the mean and standard deviation of the execution time across three runs. Also compute the mean and standard deviation of the total time to complete all jobs. Fill in the table below. Also compute the SLO violation ratio for memcached for each of the three runs; the number of datapoints with 95th percentile latency ¿ 2ms, as a fraction of the total number of datapoints.

job name	mean time [s]	std [s]
dedup	238.36	44.62
blackscholes	692.80	234.08
ferret	1187.69	239.46
freqmine	1295.99	115.36
canneal	807.90	273.39
fft	583.49	191.24
total time	1481.58	43.96

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

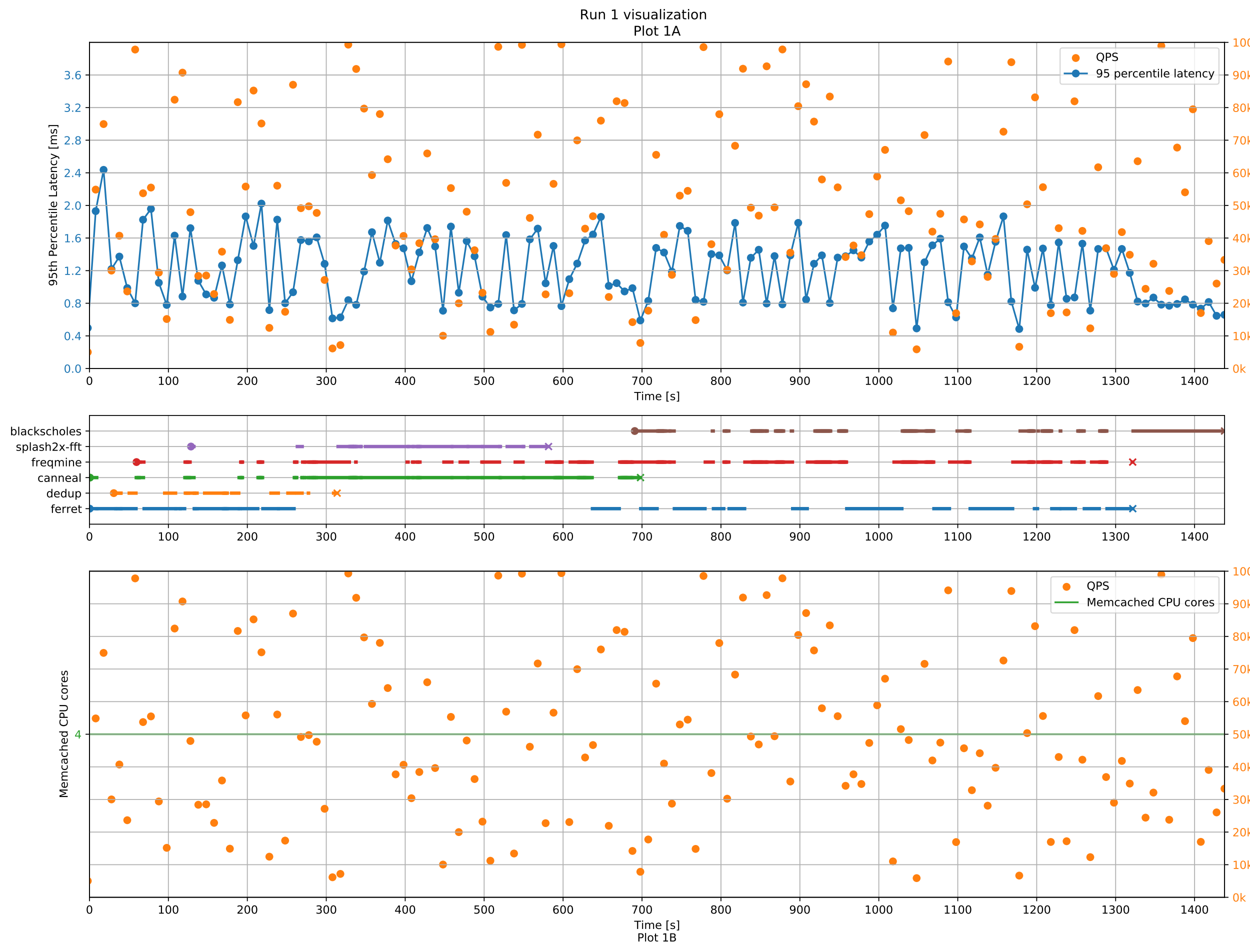


Figure 4: Dynamic scheduler trace for run 1 for Question 4. Note: for readability, instead of directly annotating the x-axis, we provided a chart presenting the executions, aligned with plots A and B in the time axis.

the x-axis and you should annotate the x-axis to indicate which PARSEC benchmark 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.

The results are presented in Figures 4, 5, and 6. The SLO violation rate for the runs 1, 2 and 3 were respectively 1.11%, 3.89% and 5%, while the mean total runtime was 24 min 42 s.

5. **[20 points]** Repeat Part 4 Question 4 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 \
```

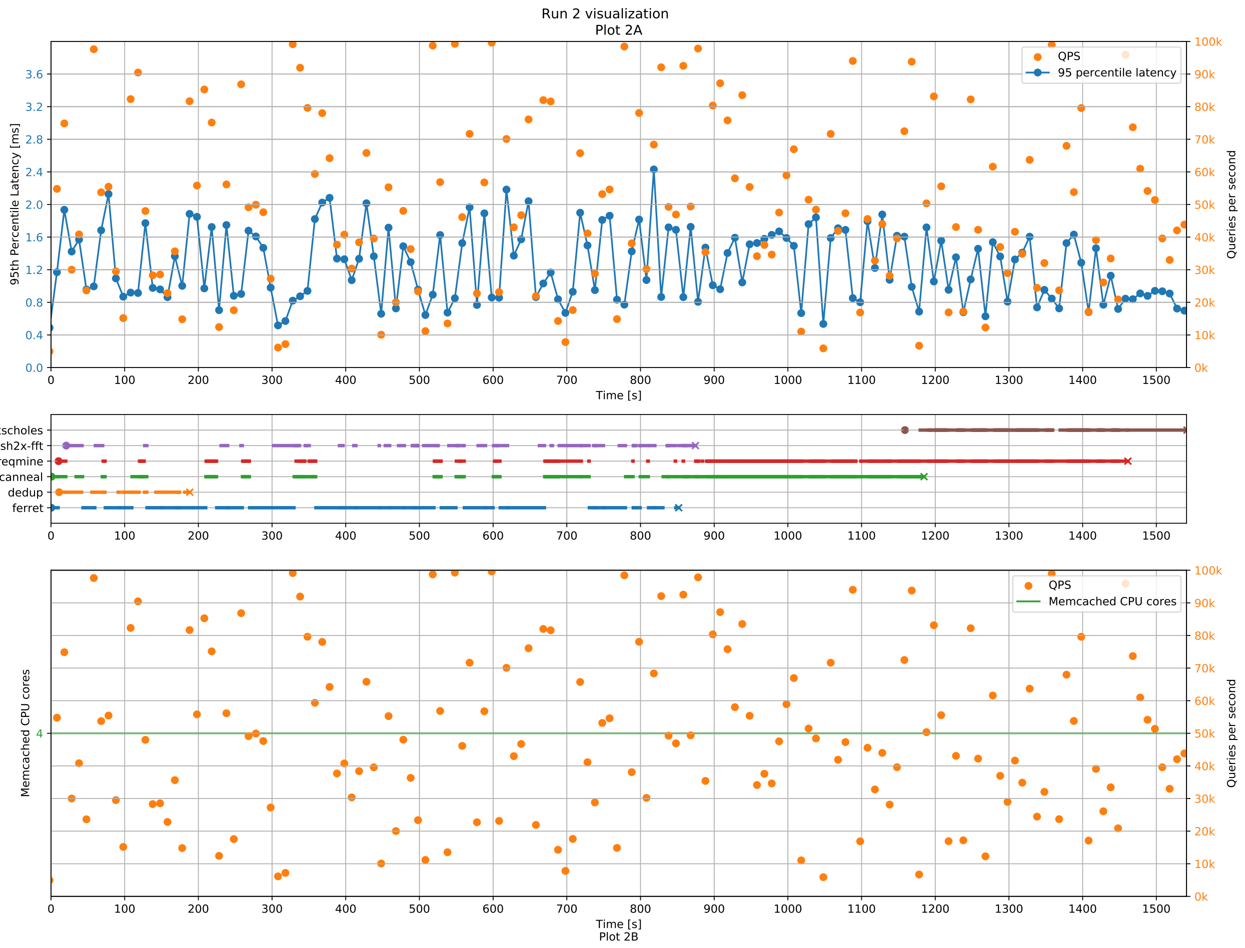


Figure 5: Dynamic scheduler trace for run 2 for Question 4. Note: for readability, instead of directly annotating the x-axis, we provided a chart presenting the executions, aligned with plots A and B in the time axis.

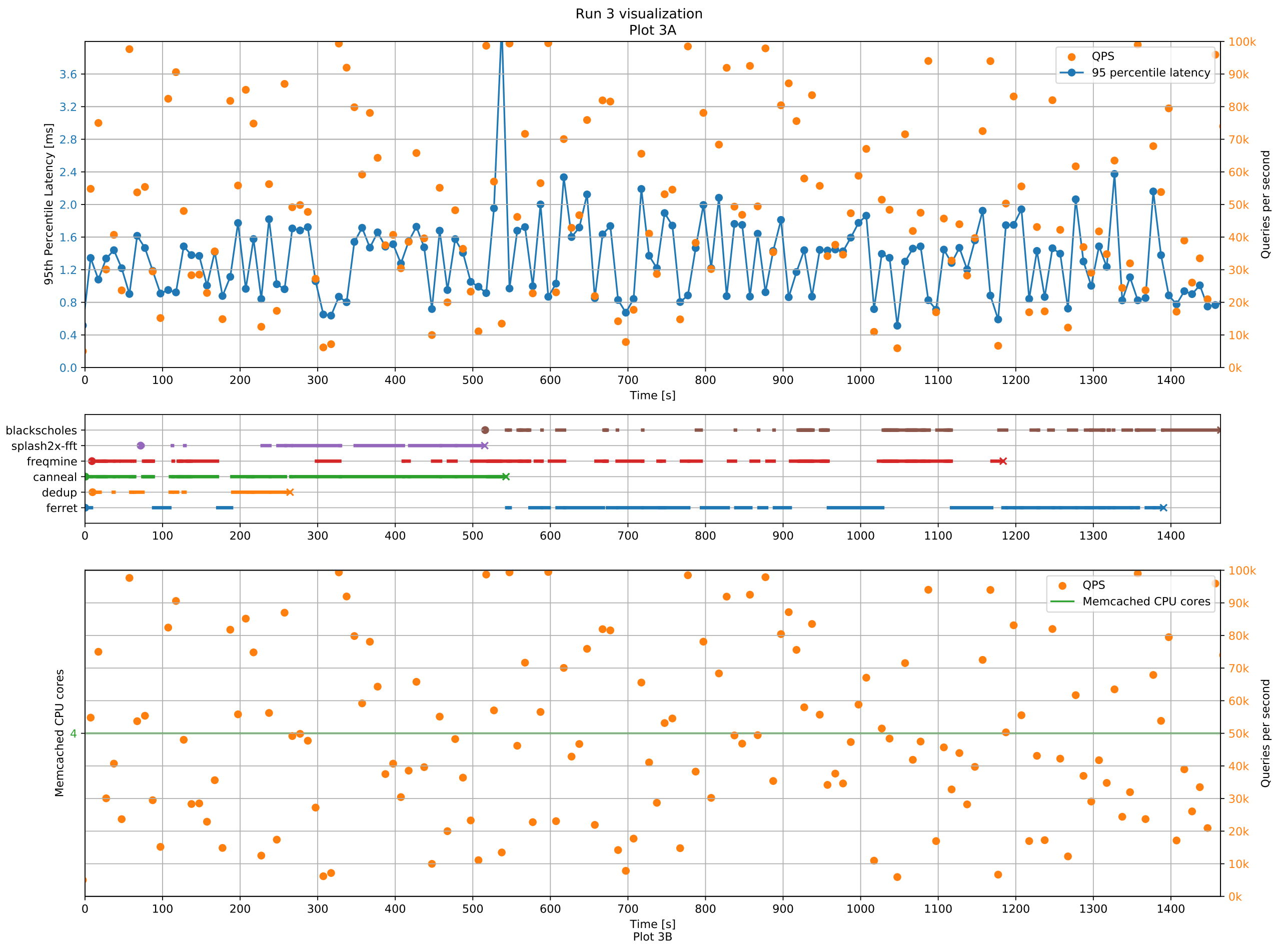


Figure 6: Dynamic scheduler trace for run 3 Question 4. Note: for readability, instead of directly annotating the x-axis, we provided a chart presenting the executions, aligned with plots A and B in the time axis.

```
--qps_interval 5 --qps_min 5000 --qps_max 100000 \
--qps_seed 42
```

You do not need to include the plots or table from Question 4 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 4. What is the SLO violation ratio for memcached (i.e., the number of datapoints with 95th percentile latency ¿ 2ms, as a fraction of the total number of datapoints) with the 5-second time interval trace?

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%? 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 4.

job name	mean time [s]	std [s]
dedup	252.50	13.61
blackscholes	853.89	50.55
ferret	1528.85	7.67
freqmine	1172.41	86.78
canneal	616.58	46.13
fft	565.45	78.37
total time	1529.32	7.61

For the 5 seconds interval, our SLO violation rate was **6.94%** and the runtime was 25 min 41 s. This is one minute slower than the average runtime with 10s interval. Also, the violation rate is much higher (6.94% vs average 3.33%). We have also observed that the container were paused more often, which suggests that memcached resource usage had higher variance.

According to our experiments, the smallest `qps_interval` which allowed us to retain the SLO violation rate below 3% was **8 seconds**. The run data is presented in Figures 7, 8, and 9. The SLO violation rate for the runs 1, 2 and 3 were respectively 0.44%, 2.67% and 1.33%. This is quite surprising given the fact that for a 10s interval we achieved higher violation rates. We believe this is due to high variance in the measurements. We observed that running the same scheduler policy during different times of day brought us results that varied by more than 5 minutes in runtime and 10 pp. in SLO violation rates.

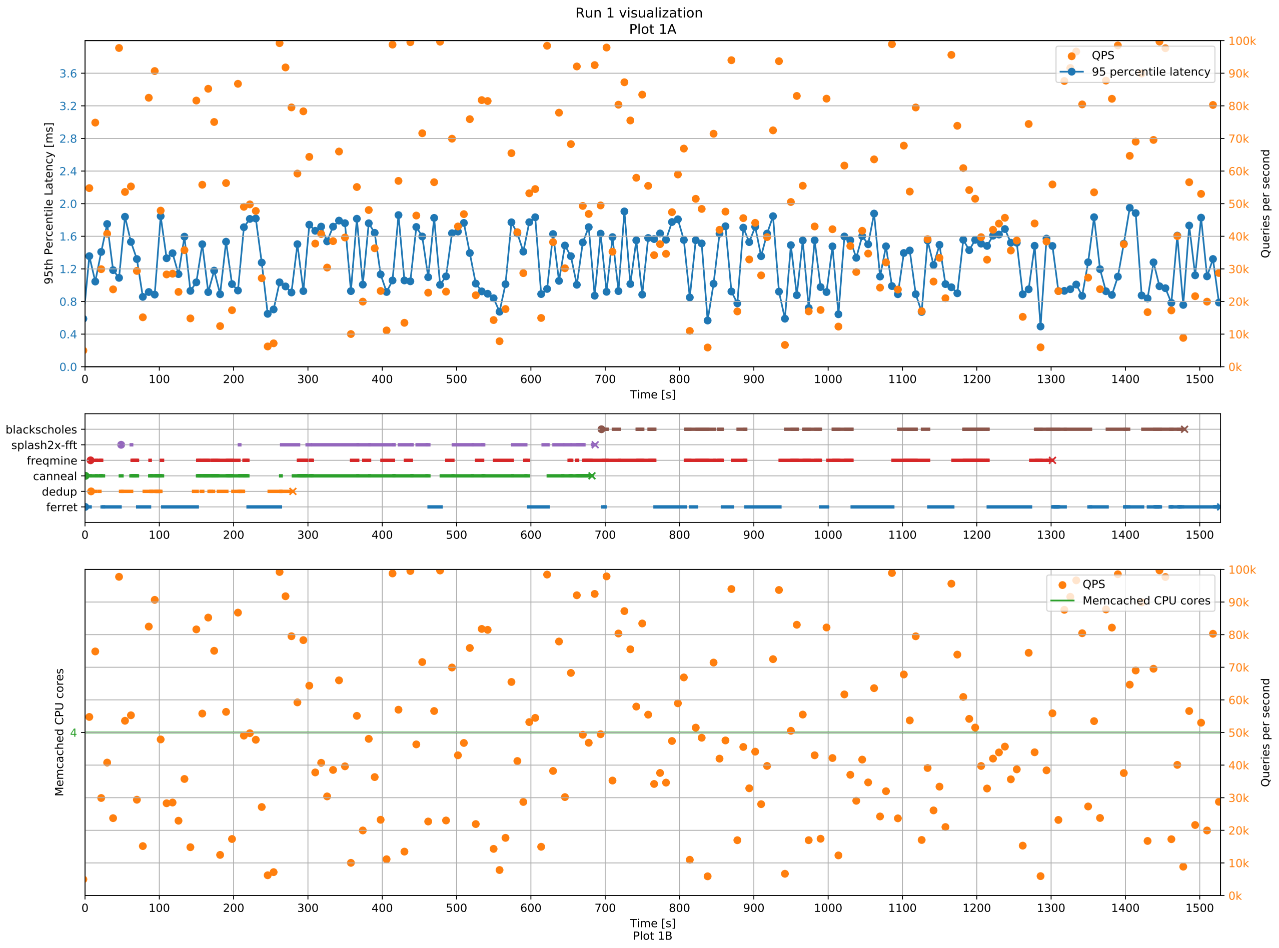


Figure 7: Dynamic scheduler trace for run 1 for Question 5 with QPS interval of 8 seconds. Note: for readability, instead of directly annotating the x-axis, we provided a chart presenting the executions, aligned with plots A and B in the time axis.

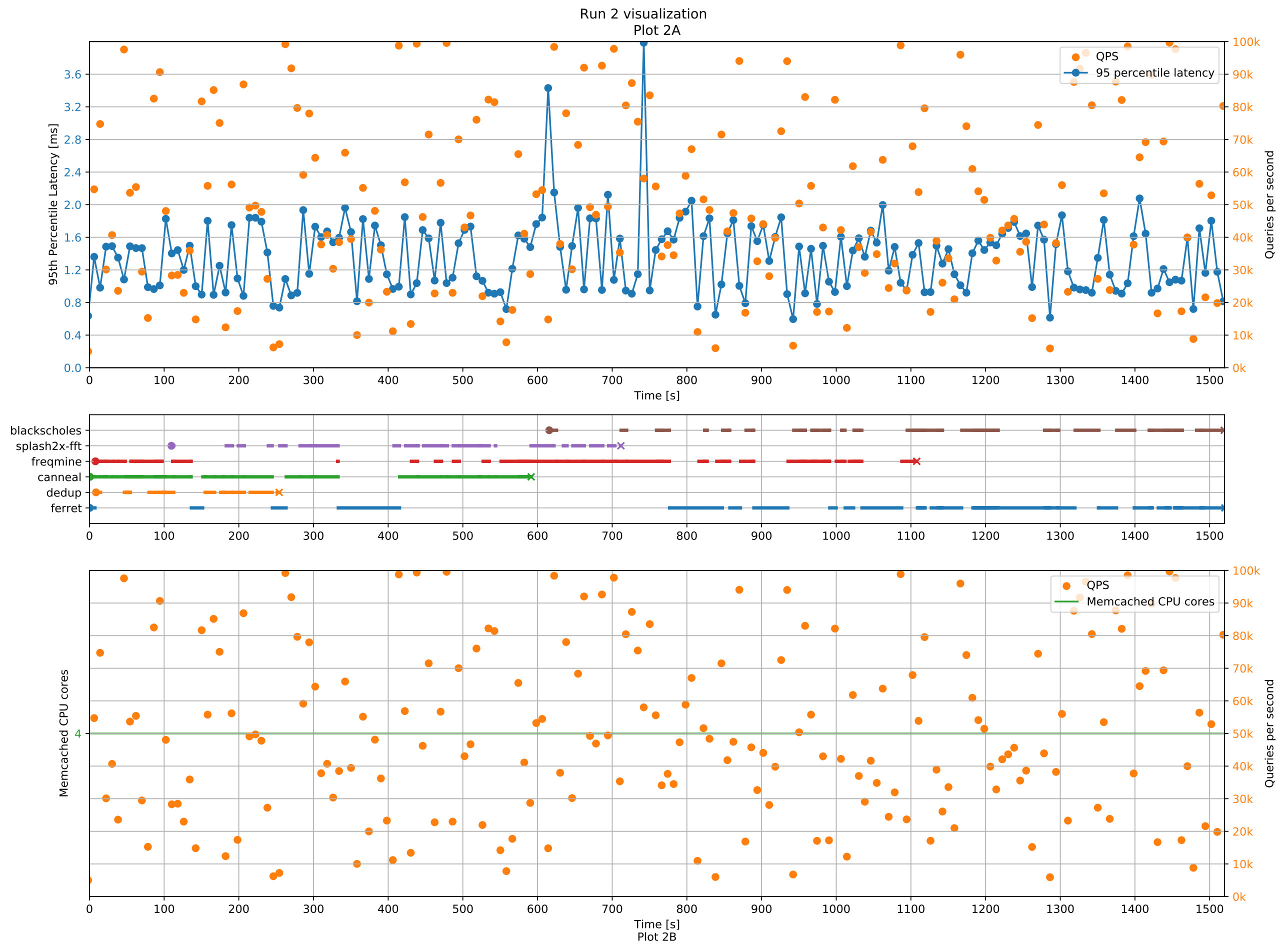


Figure 8: Dynamic scheduler trace for run 2 for Question 5 with QPS interval of 8 seconds. Note: for readability, instead of directly annotating the x-axis, we provided a chart presenting the executions, aligned with plots A and B in the time axis.

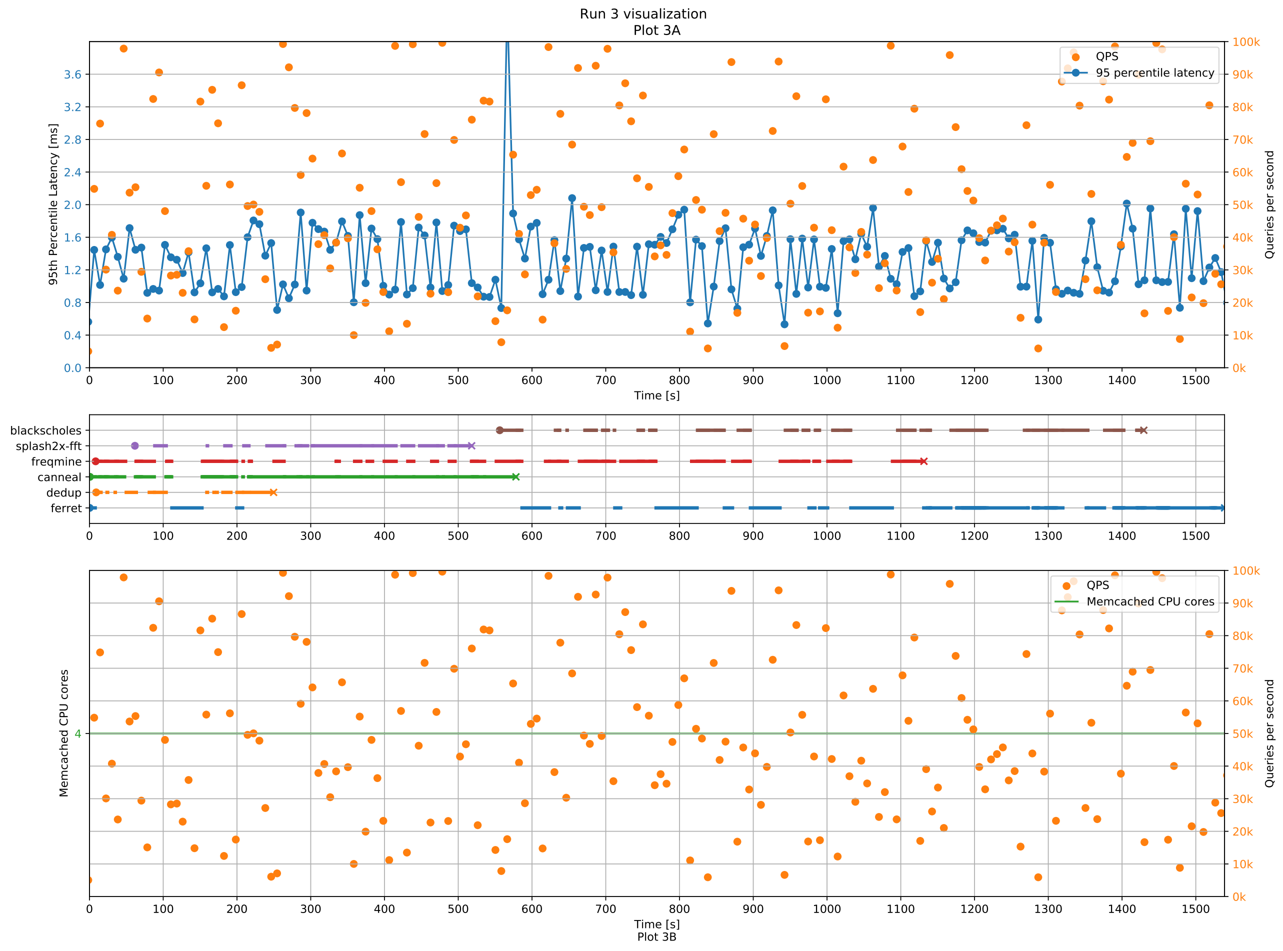


Figure 9: Dynamic scheduler trace for run 3 for Question 5 with QPS interval of 8 seconds. Note: for readability, instead of directly annotating the x-axis, we provided a chart presenting the executions, aligned with plots A and B in the time axis.