Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Systems@ETH Zürich

# Cloud Computing Architecture

Semester project report

**Group 022**
Ruben Schenk - 18-801-886
Oleh Kuzyk - 21-913-496
Thomas Gence - 18-819-979

Systems Group
Department of Computer Science
ETH Zurich
May 25, 2023

## 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 [1] 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 data points with 95th percentile latency $>$ 1ms, as a fraction of the total number of data points while the jobs are running.

   **Answer:** Since none of our data points have a 95th percentile latency $>$ 1ms, the SLO violation ratio for memcached for all three runs is 0. The plots are given in figure 1, figure 2, and figure 3. Note that the plots are aligned to $x = 0$ to the last memcached data point during which the first PARSEC job started.

   Create 3 bar plots (one for each run) of memcached p95 latency (y-axis) over time (x-axis) with annotations showing when each PARSEC job started and ended, also indicating the machine they are 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 the bar 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.

| job name | mean time [s] | std [s] |
|---|---|---|
| blackscholes | 95.67 | 0.58 |
| canneal | 153 | 7 |
| dedup | 45.67 | 0.58 |
| ferret | 157.67 | 2.31 |
| freqmine | 146.34 | 0.58 |
| radix | 35 | 0 |
| vips | 88 | 2 |
| total time | 159.67 | 2.31 |

   **Plots:**

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

   - Which node does memcached run on? Why?

     **Answer:** Memcached runs on `node-a-2core`. It runs on core 0, different from the other jobs on the same node. The reason for this is, that although it is a lightweight application, it is highly influenced by cpu and l1i interference. Placing memcached on it's own core on a node that has an even balance between processing power and memory makes sure that we respect the SLO of 1ms.

   - Which node does each of the 7 PARSEC jobs run on? Why?

     **Answer:**

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

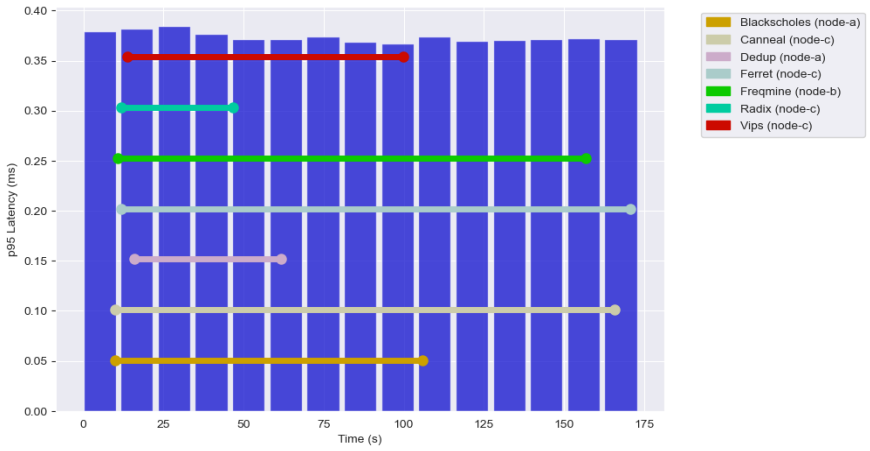


Figure 1: Run 1: 95th percentile latency of `memcached` over time.

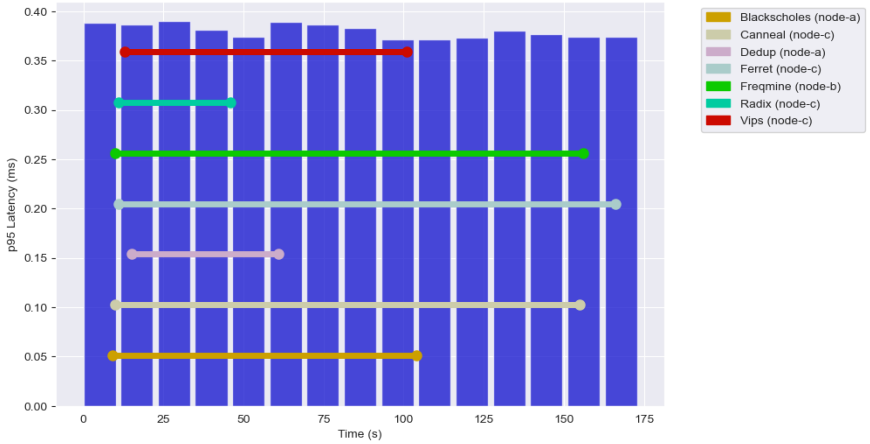


Figure 2: Run 2: 95th percentile latency of `memcached` over time.

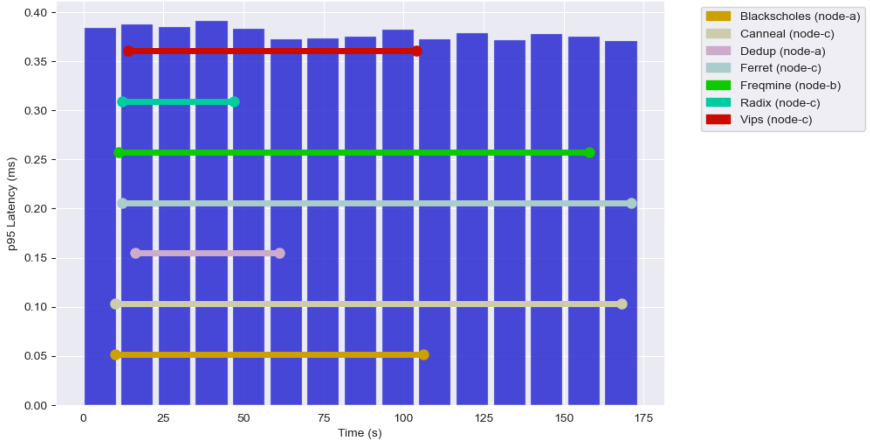


Figure 3: Run 3: 95th percentile latency of `memcached` over time.

- blackscholes: Runs on `node-a-2core` on core 1. Compared to the other PARSEC jobs it is a shorter job. Although it could profit from more cores, the total run time profits more if other longer running jobs are allocated more cores.
- canneal: Runs on `node-c-8core` on cores 4-7 together with radix and vips. Canneal is a job with a long run time and we therefore want to make use of the 4 cores. Since radix and vips are jobs with a short run time, canneal therefore has the 4 cores for its own use after radix and vips finished.
- dedup: Runs on `node-a-2core` on core 1 together with blackscholes. The reasons are the same as for blackscholes. It has a relatively short run time, optimizing its run time to the best possible time does not benefit the total run time of all jobs.
- ferret: Runs on `node-c-8core` on cores 0-3. The reasons are the same as for canneal. With a long run time we colocated it with vips, having a short run time, so that ferret has the 4 cores for its own after vips finished.
- freqmine: Runs on `node-b-4core` on cores 0-3. Freqmine is a job with one of the longest run times and we therefore want to reduce it as much as possible. As seen in part 2, it shows some speedup on 2 and 4 cores, and together with the long run time, running it on 4 cores benefits the total run time the most.
- radix: Runs on `node-c-8core` on cores 4-7 together with canneal and vips. As seen in part 2, radix shows great speedup when run on 4 cores. Secondly, it doesn't show big interference with canneal and vips, which is why we run them on the same cores.
- vips: Runs on `node-c-8core` on cores 4-7 together with canneal and radix. Similar to radix, it shows great speedup when run on 4 cores, has a relatively short run time in general, and has only medium intereference with the other two jobs on l1i.

- Which jobs run concurrently / are colocated? Why?

  **Answer:** All jobs run concurrently. The colocation, as noted in the previous question, is: memcached on `node-a-2core` core 0, dedup and blackscholes on `node-a-2core` core 1, freqmine on `node-b-4core` on cores 0-3, ferret on `node-c-8core` cores 0-3, vips, radix, and canneal on `node-c-8core` on cores 4-7.

  Dedup and blackscholes are colocated due to their already short run time and showing some of the lowest speedup when run on 2 cores amongst all the other jobs.

  Canneal, vips, and radix are colocated due to vips and radix having a short run time when run on 4 cores, leaving the rest for canneal once they are finished. They also don't show big interference among each other.

- In which order did you run 7 PARSEC jobs? Why?

  **Order (to be sorted)**: blackscholes, canneal, dedup, ferret, freqmine, radix, vips

  **Why**: The jobs are all run concurrently. The exact order in which the jobs start is determined by how long it takes to setup the containers. The ordering above is therefore ambiguous.

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

  **Answer:**

  - blackscholes: 1 thread, it is what is left on the node since we don't want to interfere with memcached.

- canneal: 4 threads, long running job, therefore benefiting a lot in terms of total runtime.
- dedup: 1 thread, it is what is left on the node since we don't want to interfere with memcached.
- ferret: 4 threads, only job that is on the node, therefore allocated to all the resources available.
- freqmine: 4 threads, gets assigned half of the resources of the node since it is a long running job.
- radix: 4 threads, shows great speedup on 4 threads, finishing quickly, therefore leaving the resources to canneal afterwards.
- vips: 4 threads, shows great speedup on 4 threads, finishing quickly, therefore leaving the resources to canneal afterwards.

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

  **Answer:** We modified the yaml files of all PARSEC jobs to include the taskset command which allows us to specify the cores the job runs on and the number of threads. We also added the `nodeSelector` in the yaml files of the jobs and memcached to set the node the jobs run on. We used the `kubectl get jobs` function to check which jobs are still running, allowing us to automate the whole process with a shell-script.

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

  **Answer:** Our goal was to first identify the longest running jobs, those included ferret, freqmine, and canneal. We then assigned these jobs to their respective nodes/cores to reduce the total time it takes to finish all jobs. The remaing jobs were distributed to the remaining cores. We colocated jobs that show low interference between each other. This also meant some trade-offs: Dedup and blackscholes could run much faster when run on 2 cores on their own, but interfere with other jobs too much. Similarly for vips and radix when run on 4 cores on their own, but the time saved is small compared to ferret, freqmine, and canneal. Assigning the jobs specific cores allowed us to make sure there is the best use of resources at any time. Specifically, with freqmine, ferret, and canneal all running on 4 separate cores and taking about the same amount of time to finish, we can make sure that those resources are utilized to the maximum. The remaining two cores are used by memcached, dedup, and blackscholes, again maximizing the resource utilization of those cores.

Please attach your modified/added YAML files, run scripts, experiment outputs and report as a zip 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 [76 points]

1. **[20 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) [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

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:** We average across 3 runs and calculated error bars as average standard deviation.

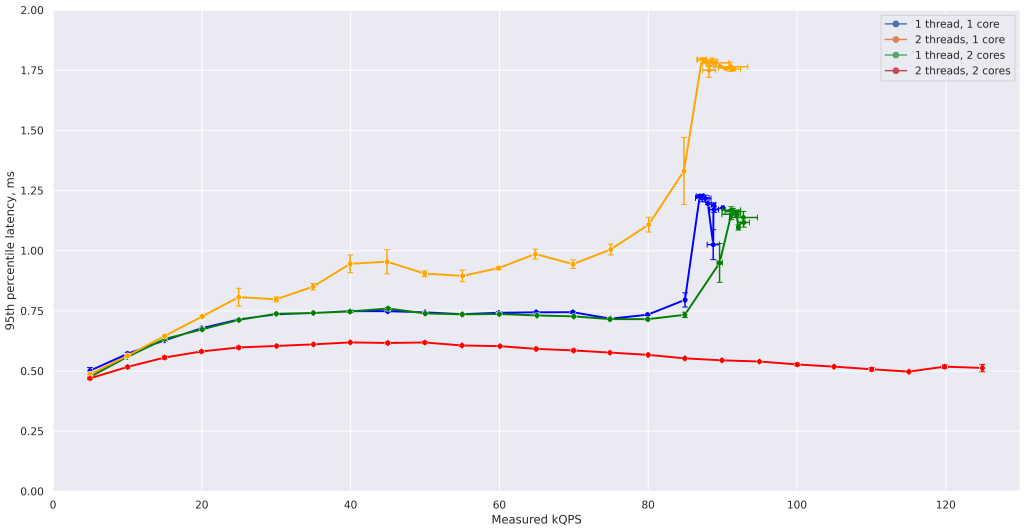


Figure 4: 95th percentile latency vs. QPS

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:** As we can see from the plot, running on 1 thread is almost the same independent of the number of cores used, because at every moment only one core is utilized. In case of 2

threads and 1 core, threads compete for using the core, thus, we have the highest increase in the latency value. Having 2 threads and 2 cores seems to be optimal, since the latency values are almost constant, because resources can be allocated in a better way.

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:** The only configuration that supports the whole range of QPS without violating the 1ms latency SLO is $T = 2$ and $C = 2$.

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:** Using $T = 2$, we can guarantee the 1ms 95th percentile latency SLO switching from $C = 1$ to $C = 2$ when load increases and there is a risk to violate the latency SLO.

d) [7 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 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:** Plots of 95th percentile latency and CPU utilization with $T = 2$ and $C = 1$ and $C = 2$ values are in Figures 5 and 6

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 PARSEC benchmarks on the 4-core VM. The goal of your scheduling policy is to successfully complete all PARSEC

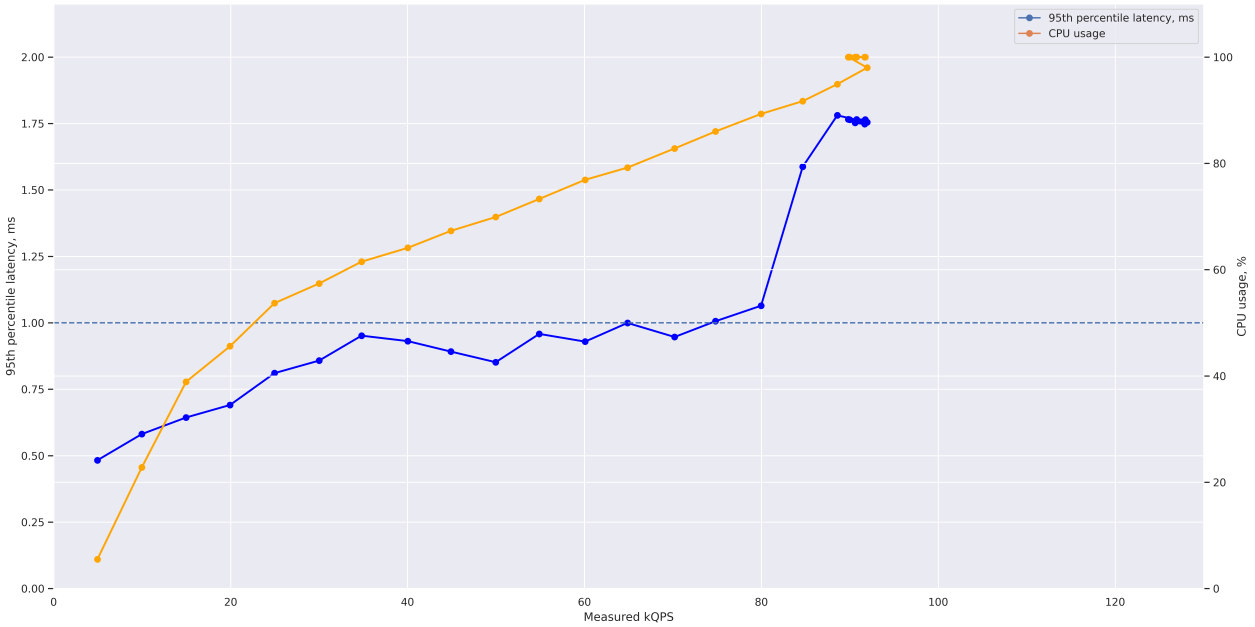


Figure 5: 95th percentile latency and CPU utilization with $C = 1$ and $T = 2$

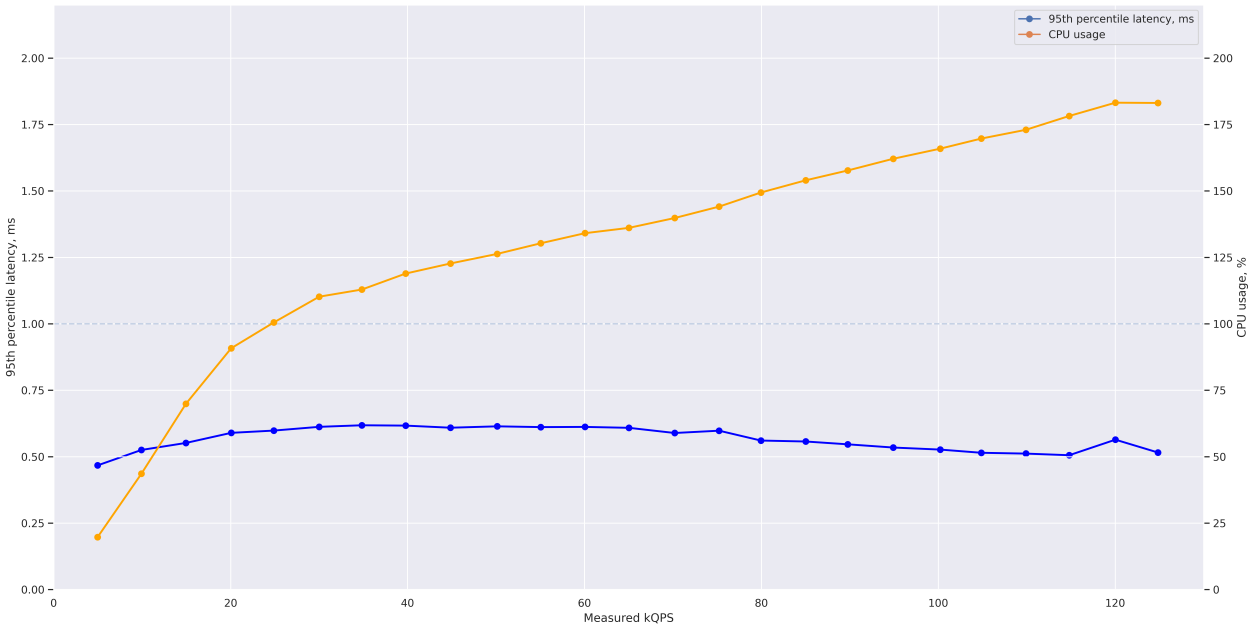


Figure 6: 95th percentile latency and CPU utilization with $C = 2$ and $T = 2$

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 PARSEC 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 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.

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

  **Answer:** We create two classes of jobs: "long" jobs and "short" jobs, based on how heavy they are. They run on separate cores and do not interfere. We also assign each jobs a specific number of threads based on how well they parallelize. MEMCACHED starts running on 1 core and as soon as it reaches 40% CPU utilization, there is a risk of SLO violation, and therefore we switch MEMCACHED to use two cores and pause the jobs that run on the core we switched MEMCACHED to. The other way around, we switch from 2 cores to 1 core and resume the paused jobs in case total CPU utilization on MEMCACHED cores is less than 35%. If all the "long" jobs finish, we can start remaning "short" jobs on the freed cores.

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

  **Answer:** As soon as the 95th percentile latency approaches 1ms (which happens at around 40% CPU usage), we switch from 1 to 2 cores. The other way round, if the load is too low on 2 cores (35% in total), we go back to using only one core.

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

  **Answer:** Long-executing jobs were assigned 2 cores, while short-running jobs only 1 core. For the long-running jobs we dedicated cores 2 and 3, while for short-running ones we used core 1.

  - blackscholes: 1 core.
  - canneal: 2 cores.
  - dedup: 1 core.
  - ferret: 2 cores.
  - freqmine: 2 cores.
  - radix: 1 core.
  - vips: 1 core.

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

  **Answer:** For most of the jobs, we used 4 threads, because in most cases they are quite well parallelizable. For FERRET and FREQMINE we selected 2 threads since their speedup is not noticeable from 2 to 4 threads (as seen in part 2 of the project).

  - blackscholes: 4 thread.
  - canneal: 4 threads.
  - dedup: 4 thread.

  - ferret: 2 threads.
  - freqmine: 2 threads.
  - radix: 4 threads.
  - vips: 4 threads.

- Which jobs run concurrently / are collocated and on which cores? Why?
  **Answer:** All the "long" jobs (FREQMINE, FERRET, BLACKSCHOLES, and CANNEAL) are running on cores 2 and 3 one after another. All the other "short" jobs are run one after the other on core 1. MEMCACHED runs on core 0 or cores 0 and 1. The main reason behind such allocation was to separate heavy jobs from MEMCACHED and to isolate MEMCACHED from other jobs if possible. Lighter jobs can be paused if MEMCACHED needs to be extended, to two cores.

- In which order did you run the PARSEC jobs? Why?
  **Order** (to be sorted): canneal, vips, radix, ferret, dedup, freqmine, blackscholes
  **Why:** As already mentioned we have two lists of jobs, the long jobs which take more time and are more compute-heavy which we run on cores 2 and 3. These are (in this order): [canneal, ferret, freqmine, blackscholes]. Then we have a list of shorter, less heavier jobs. These are (in this order): [vips, radix, dedup].

- How does your policy differ from the policy in Part 3? Why?
  **Answer:** The main difference in the problem statement is the availability of resources. In comparison to the previous task, we have only 1 VM with 4 cores available, therefore, we can not start all of the jobs simultaneously without violating latency requirements and fitting in the CPU. This way, we need to pause some of the jobs to give some resources for MEMCACHED in case there is a risk of violation of SLO and also run jobs on the same cores one after the other.

- How did you implement your policy? e.g., docker cpu-set updates, taskset updates for memcached, pausing/unpausing containers, etc.
  **Answer:** In our scheduling policy, we keep job parameters (threads, cores) at their optimal values while simply pausing/unpausing while keeping track of the CPU utilization.

- Describe the design choices, ideas and trade-offs you took into account while creating your scheduler (if not already mentioned above):
  **Answer:** The main trade-off we make is between cpu resources for memcached and for short jobs. We prioritize cpu resource availability for memcached to respect the SLO as good as possible, pausing the running short job if memcached needs more resources.

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 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. 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 data points with 95th percentile latency $>$ 1ms, as a fraction of the total number of datapoints. You should only report the runtime, excluding time spans during which the container is paused.

**Answer:** The SLO violations for the three runs are as follows:

- Run 1: $\frac{4}{93} = 0.04301 = 4.301\%$
- Run 2: $\frac{7}{93} = 0.07526 = 7.526\%$
- Run 3: $\frac{6}{95} = 0.06315 = 6.315\%$

| job name | mean time [s] | std [s] |
|---|---|---|
| blackscholes | 69 | 1 |
| canneal | 188 | 3 |
| dedup | 44 | 15 |
| ferret | 248 | 1 |
| freqmine | 395 | 5 |
| radix | 71 | 2 |
| vips | 145 | 4 |
| total time | 921 | 12 |

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 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. For the plot, use the colors proposed in this template (you can find them in `main.tex`).

**Plots:** Corresponding plots are in Figures 7, 8, 9, 10, 11, and 12.

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 \
```

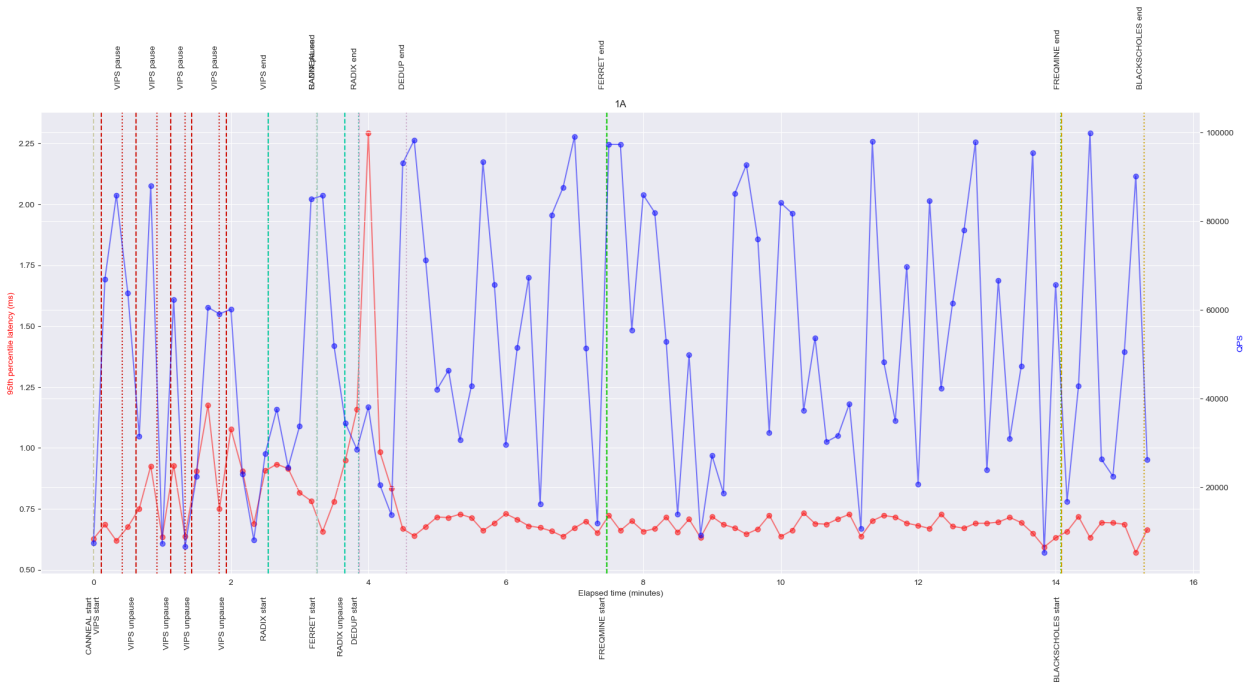


Figure 7: Plot 1A: 95th percentile latency and QPS.

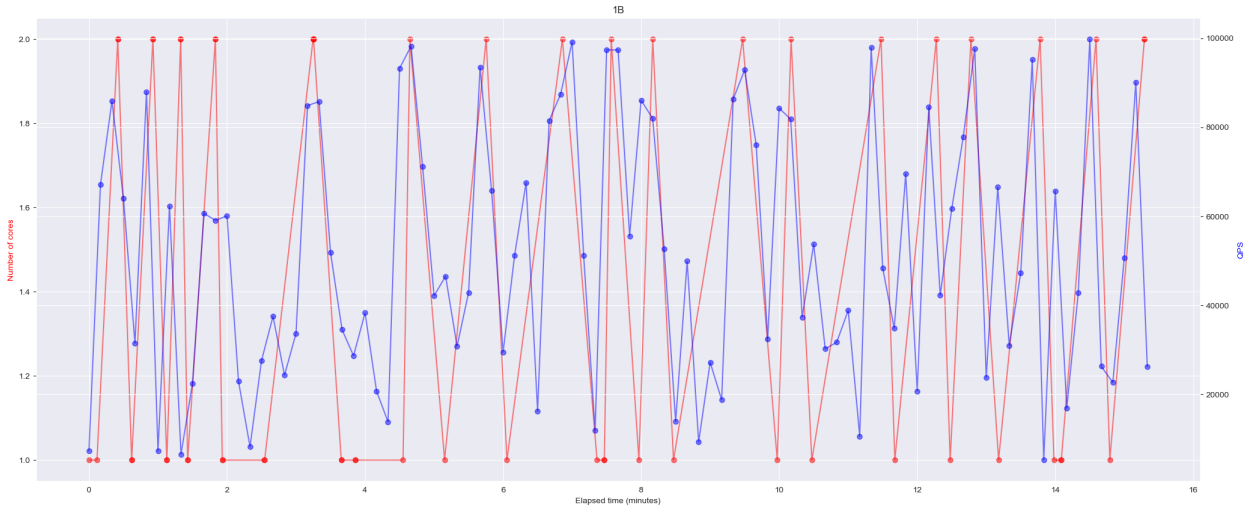


Figure 8: Plot 1B: Number of cores of MEMCACHED and QPS.

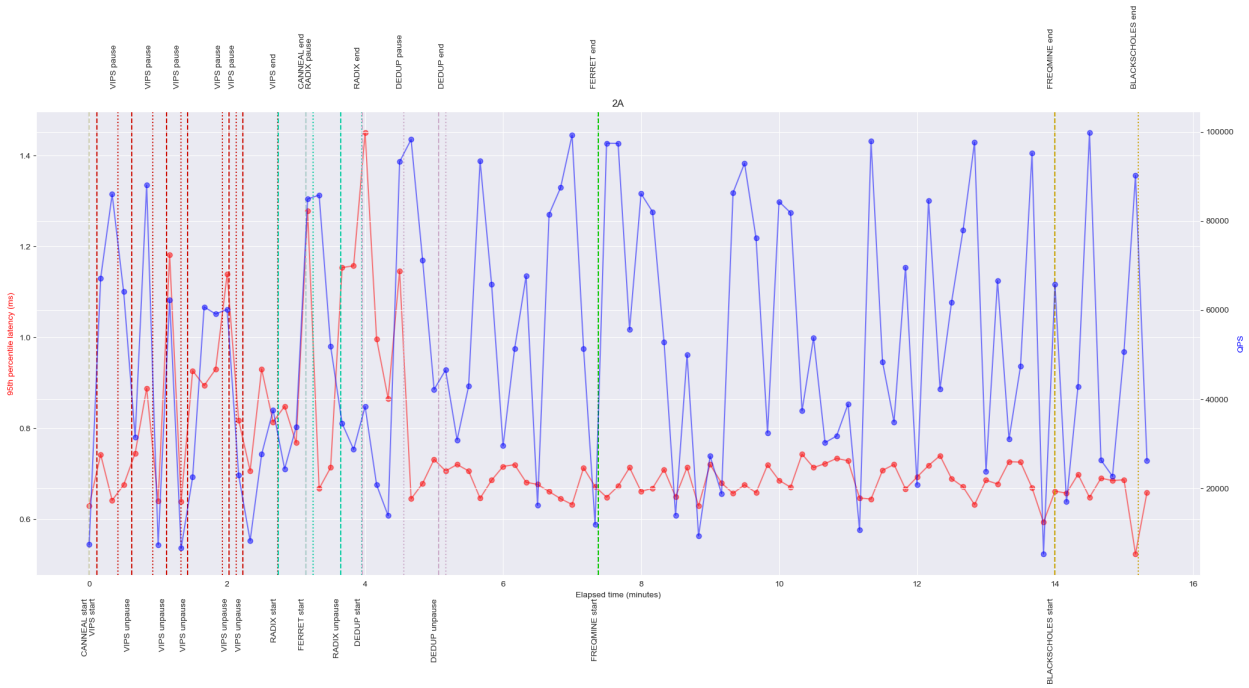


Figure 9: Plot 2A: 95th percentile latency and QPS.

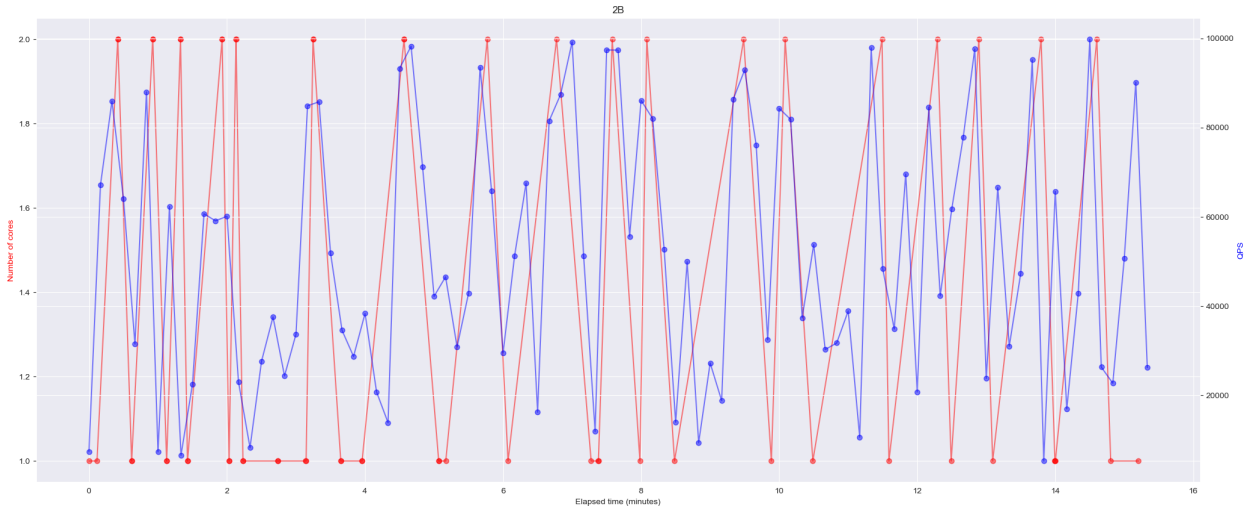


Figure 10: Plot 2B: Number of cores of MEMCACHED and QPS.

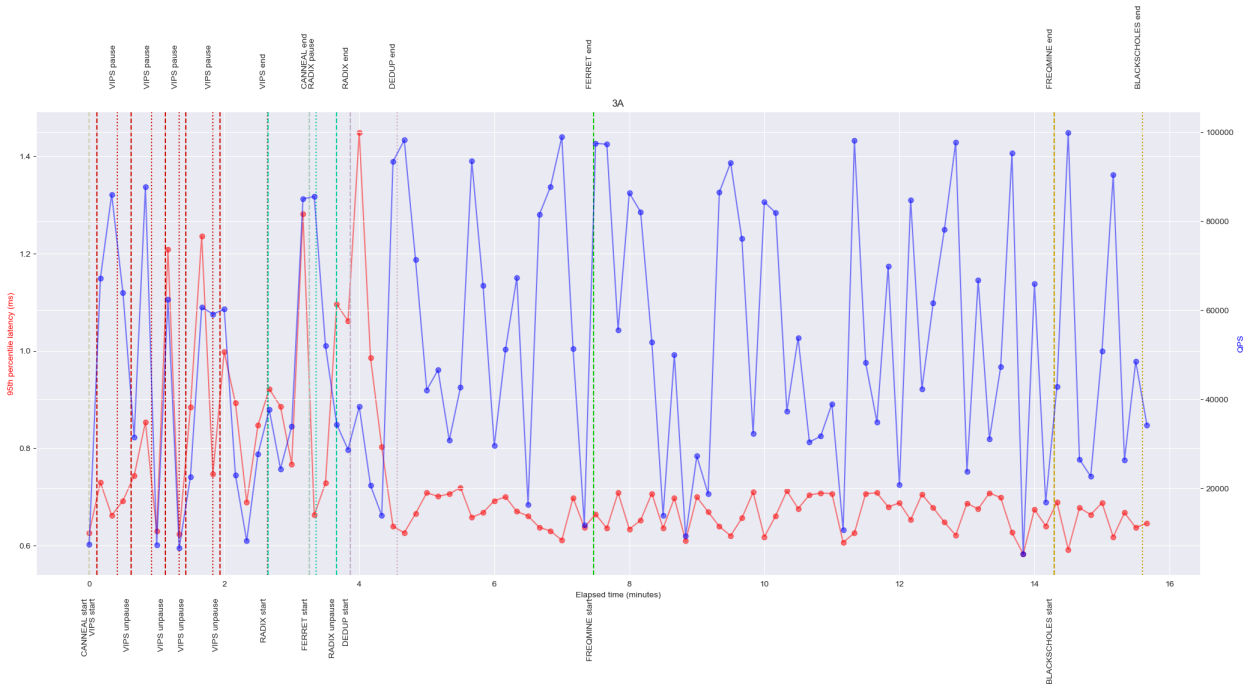


Figure 11: Plot 3A: 95th percentile latency and QPS.

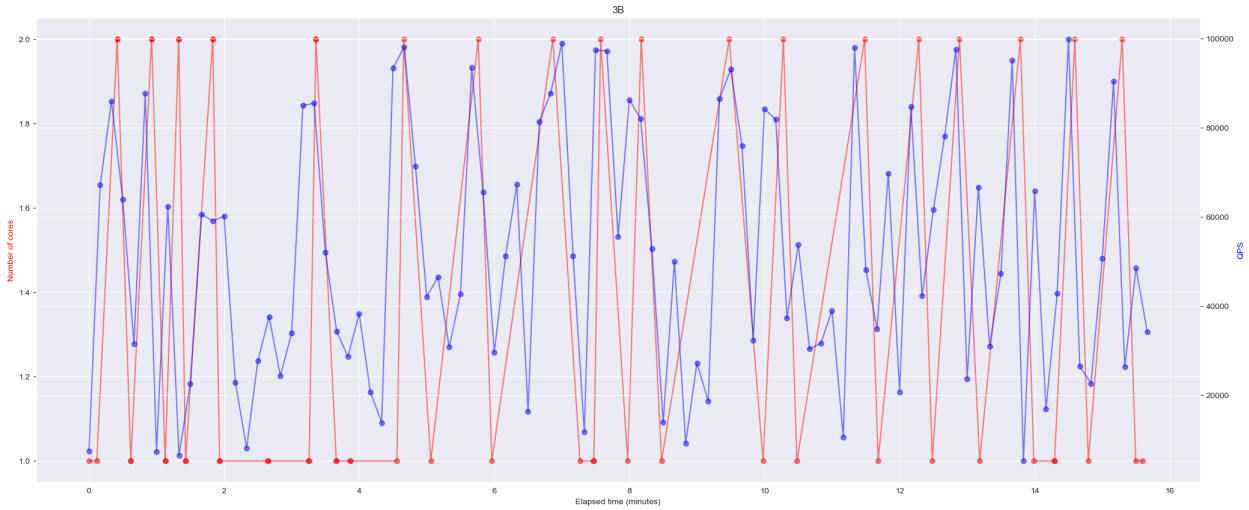


Figure 12: Plot 3B: Number of cores of MEMCACHED and QPS.

```
--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:** The total run time is a little lower than before (894s compared to 921s). However, resulting from this, due to less core updates on memcached, there is a higher SLO violation ratio (0.24731 compared to 0.06049). Since our controller checks the cpu usages every 6s, it can miss or react late to increased QPS if the interval is at 5s.

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?

**Answer:** $\frac{69}{279} = 0.24731 = 24.731\%$

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 `qps_interval` that allows our controller to respond fast enough to keep SLO violation ratio under 3% is 9.

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

**Answer:** The main feature that affects this is the cpu usage check we perform on the core(s) memcached runs on and how often we do them. In our controller we perform such a check every 6 seconds. For QPS intervals lower than 9s there is a high chance that we miss or react too late to an increased load on the memcached core(s), resulting in more SLO violations.

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.

**Plots:**

| job name | mean time [s] | std [s] |
|---|---|---|
| blackscholes | 70.1 | 2.6 |
| canneal | 192.9 | 4.1 |
| dedup | 48.9 | 15.6 |
| ferret | 315.5 | 118.9 |
| freqmine | 412.5 | 42.4 |
| radix | 59.5 | 10.9 |
| vips | 124.3 | 8.5 |
| total time | 923 | 10.8 |

Corresponding 1A, 1B, 2A, 2B, 3A, AND 3B plots are in Figures 13, 14, 15, 16, 17, and 18.

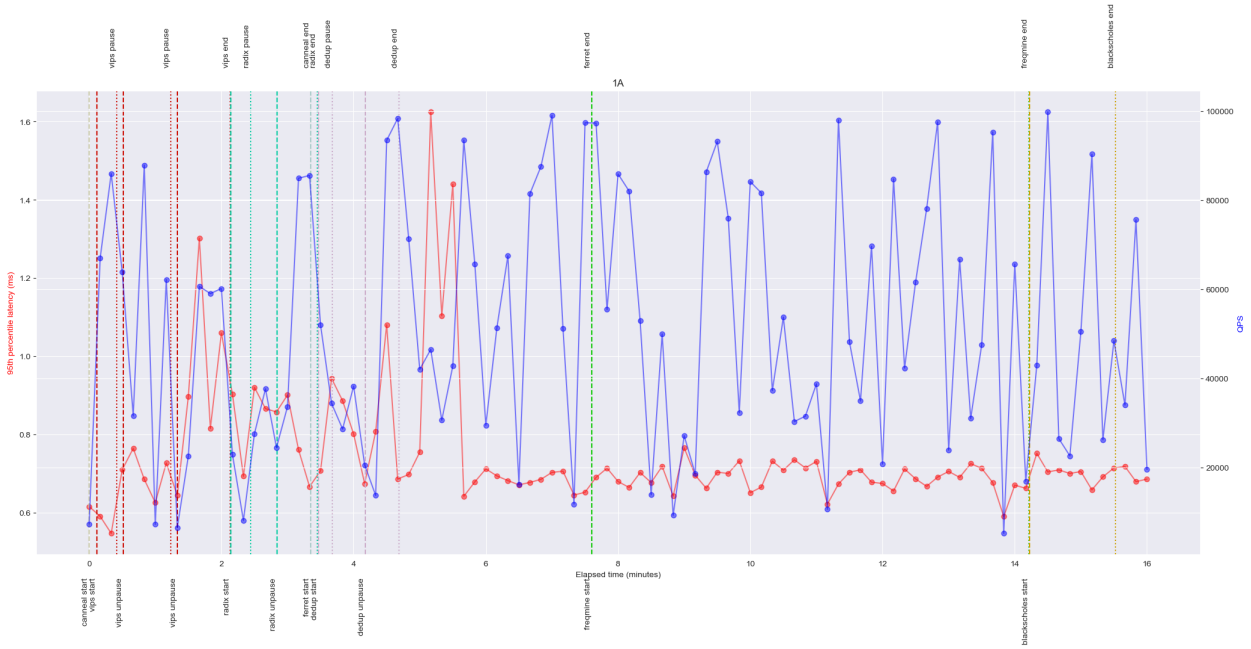


Figure 13: Plot 1A: 95th percentile latency and QPS.

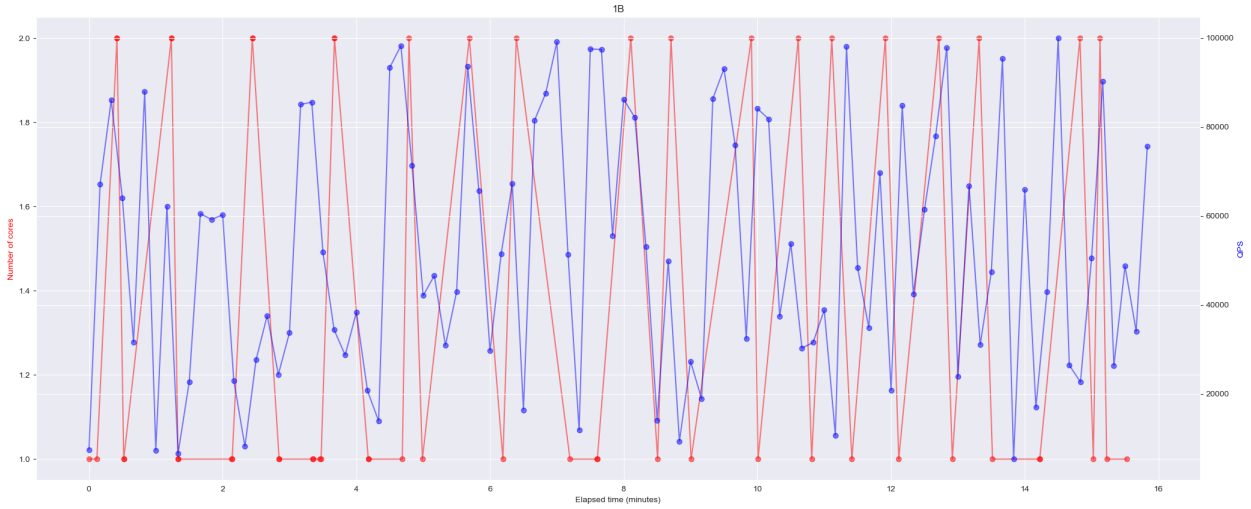


Figure 14: Plot 1B: Number of cores of MEMCACHED and QPS.

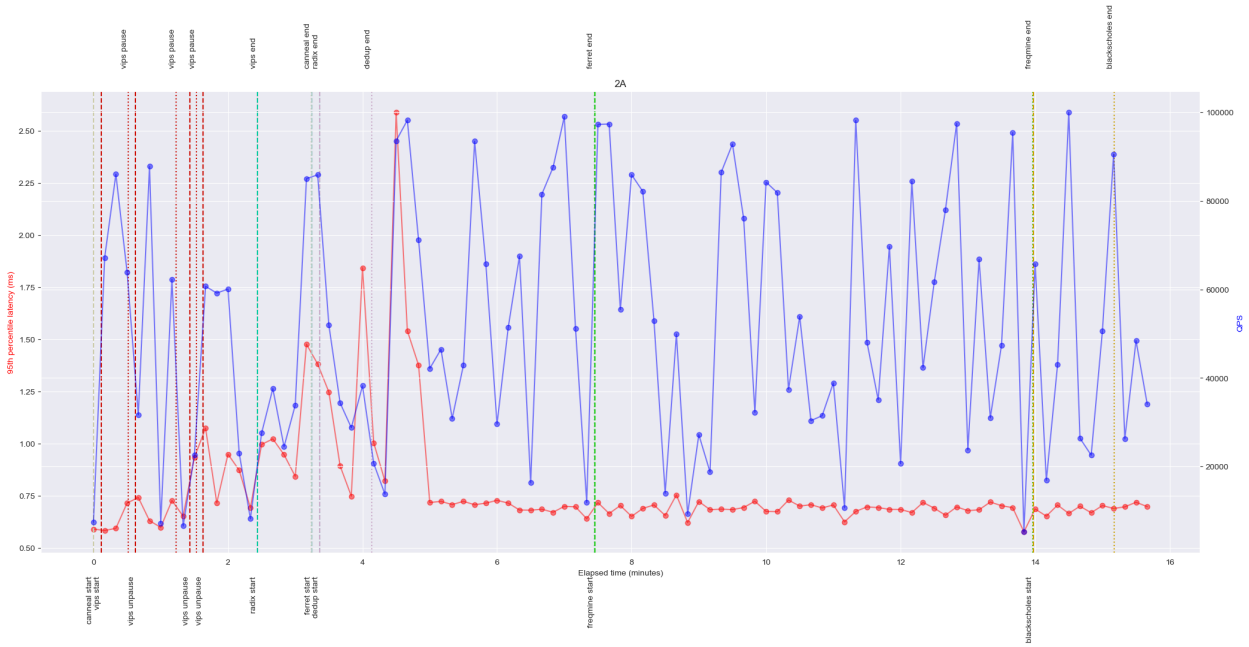


Figure 15: Plot 2A: 95th percentile latency and QPS.

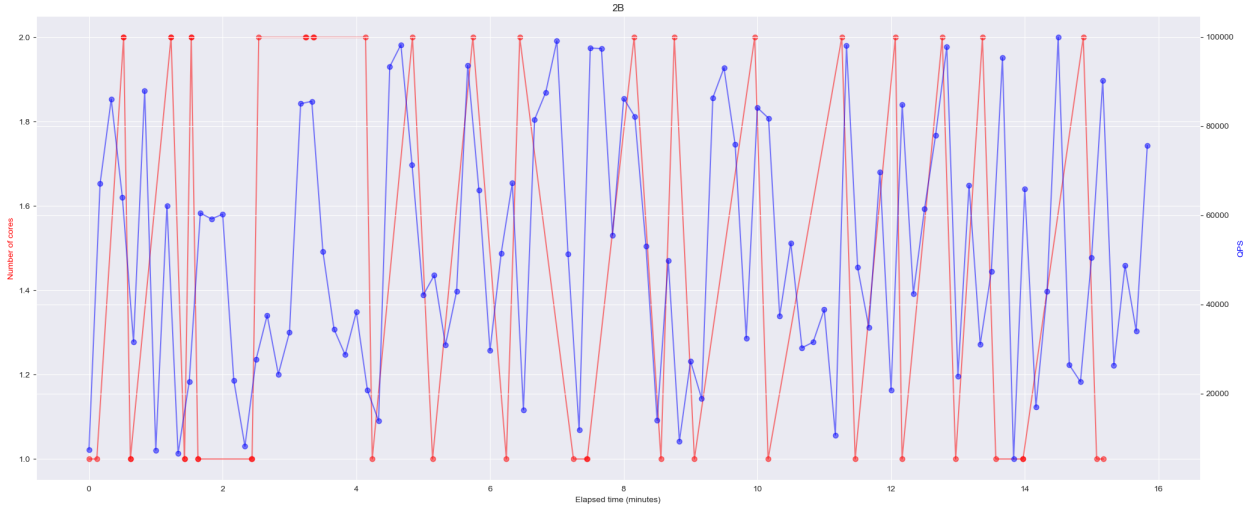


Figure 16: Plot 2B: Number of cores of MEMCACHED and QPS.

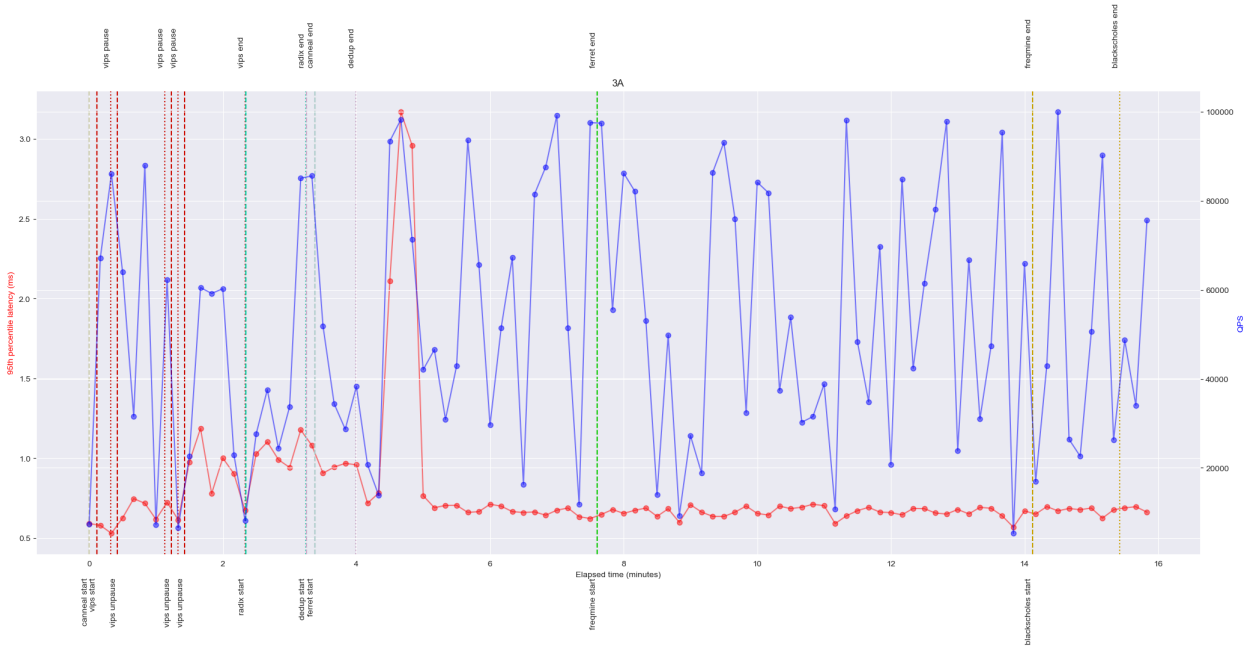


Figure 17: Plot 3A: 95th percentile latency and QPS.

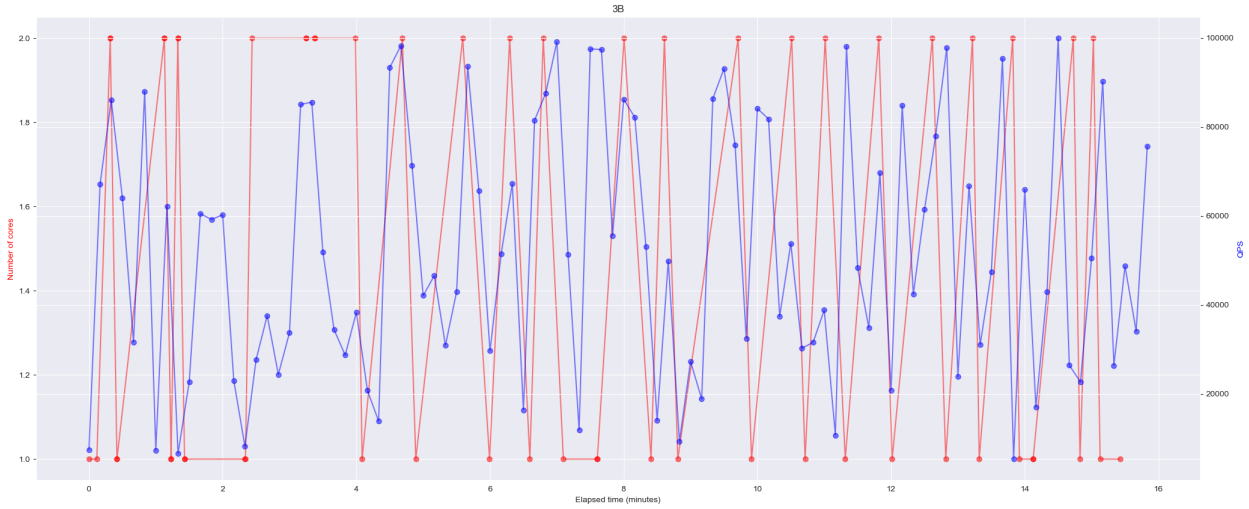


Figure 18: Plot 3B: Number of cores of MEMCACHED and QPS.