Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Cloud Computing Architecture

Semester project report

Group 054
Pitcho Oscar - 17-809-971
Perone Luca - 18-812-388
Oester Robin - 19-932-557

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: SLO violation ratio for memcached across the three runs is 0%.

   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	124	20.99
canneal	140	12.96
dedup	67.67	13.30
ferret	151.67	3.40
freqmine	150	7.07
radix	86	24.26
vips	75.67	6.94
total time	158.67	7.13

   Plots:
   Figures 1, 2 and 3

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

   - Which node does memcached run on? Why?

     Answer: We run memcached on core 1 of node-a for two reasons: First, running memcached on a single core with a single thread without any same-core collocation is enough to be below the SLO - hence there is no need to add parallelism to memcached. Second, so that other jobs may benefit from parallelism while minimizing the risk of interference with memcached, we run it on the machine with the fewest cores. We therefore reserve one core for memcached as part 1 has shown its sensitivity to CPU and L1 instruction cache interference, both of which would yield an increased performance risk if memcached were to share its core.

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

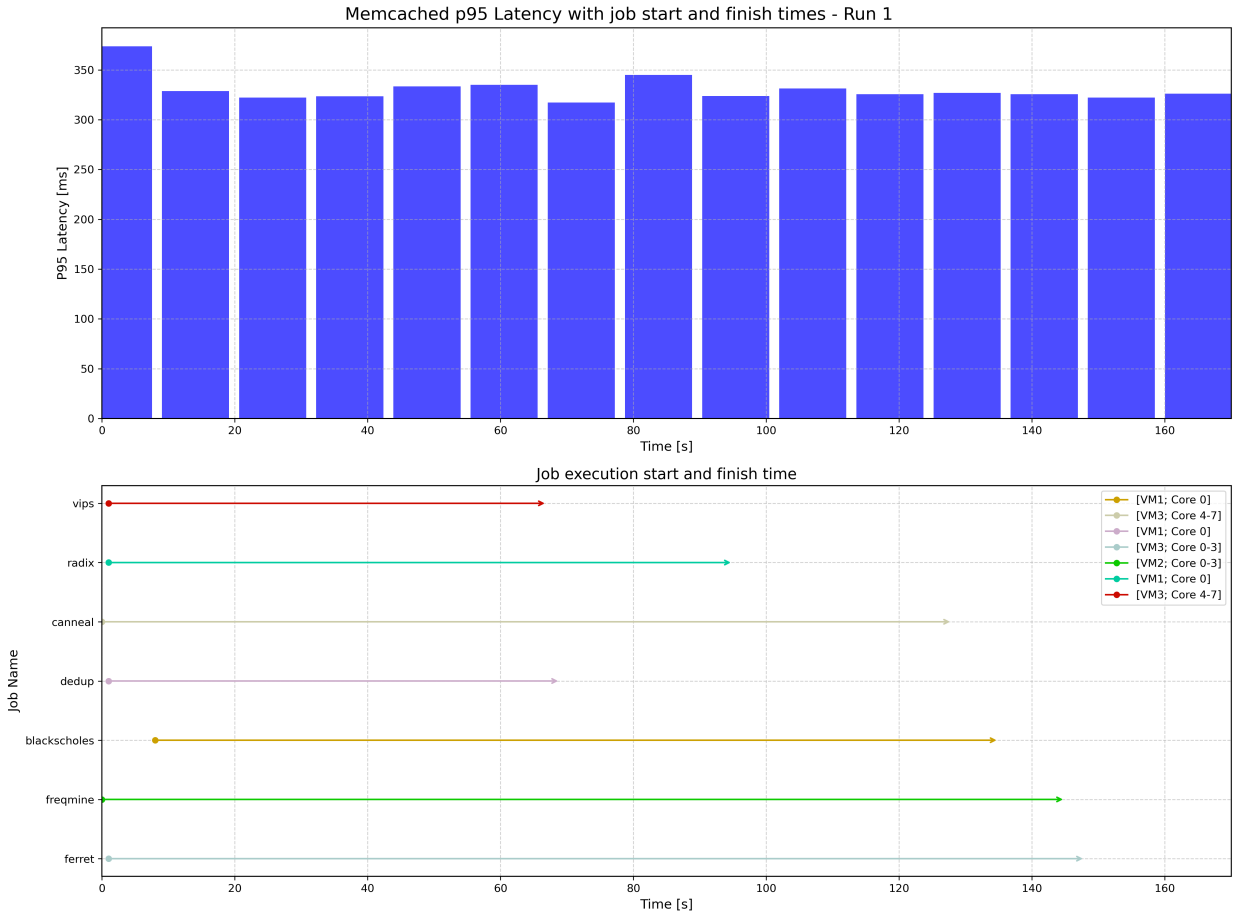

Figure 1: Evolution of p95 latency over the lifetime of parsec jobs - Run 1

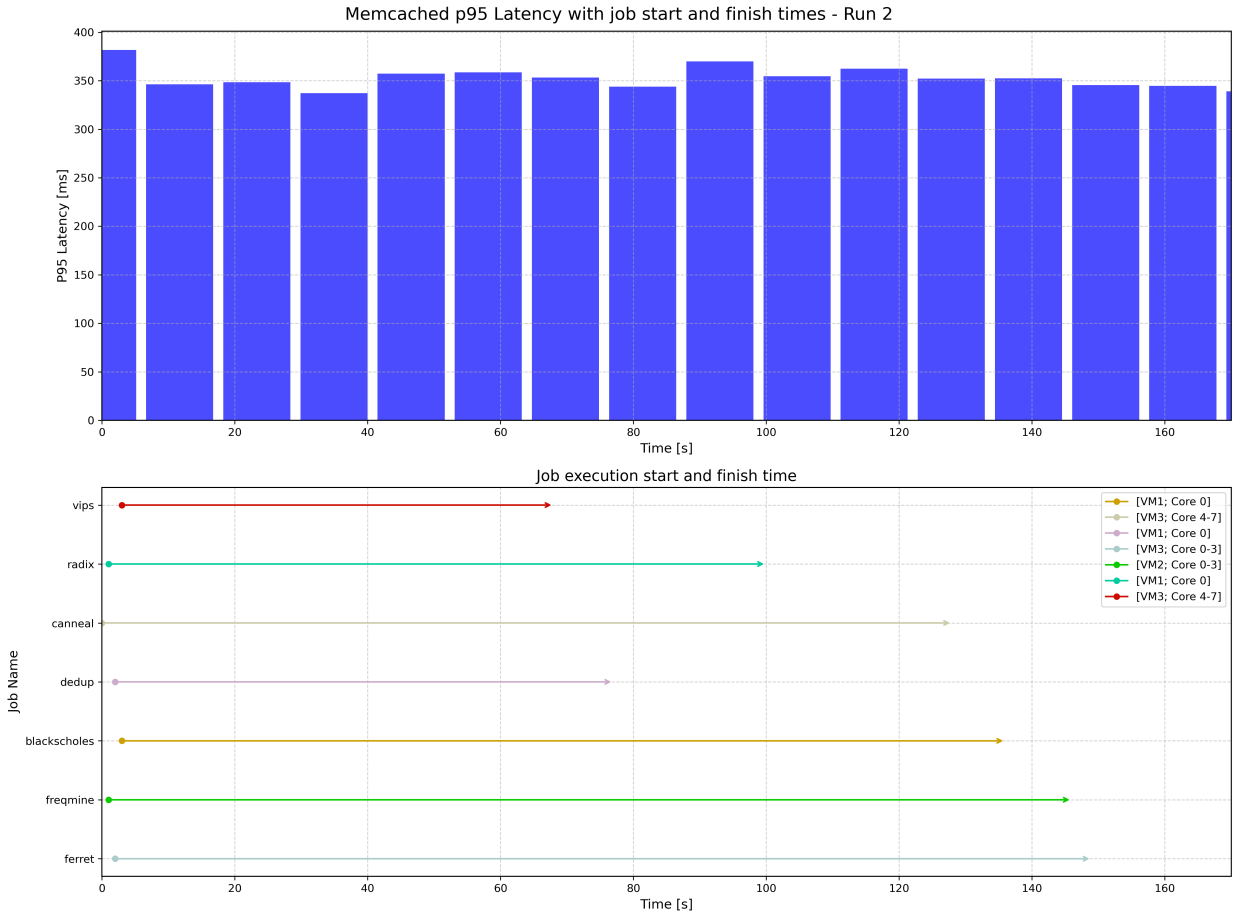

Figure 2: Evolution of p95 latency over the lifetime of parsec jobs - Run 2

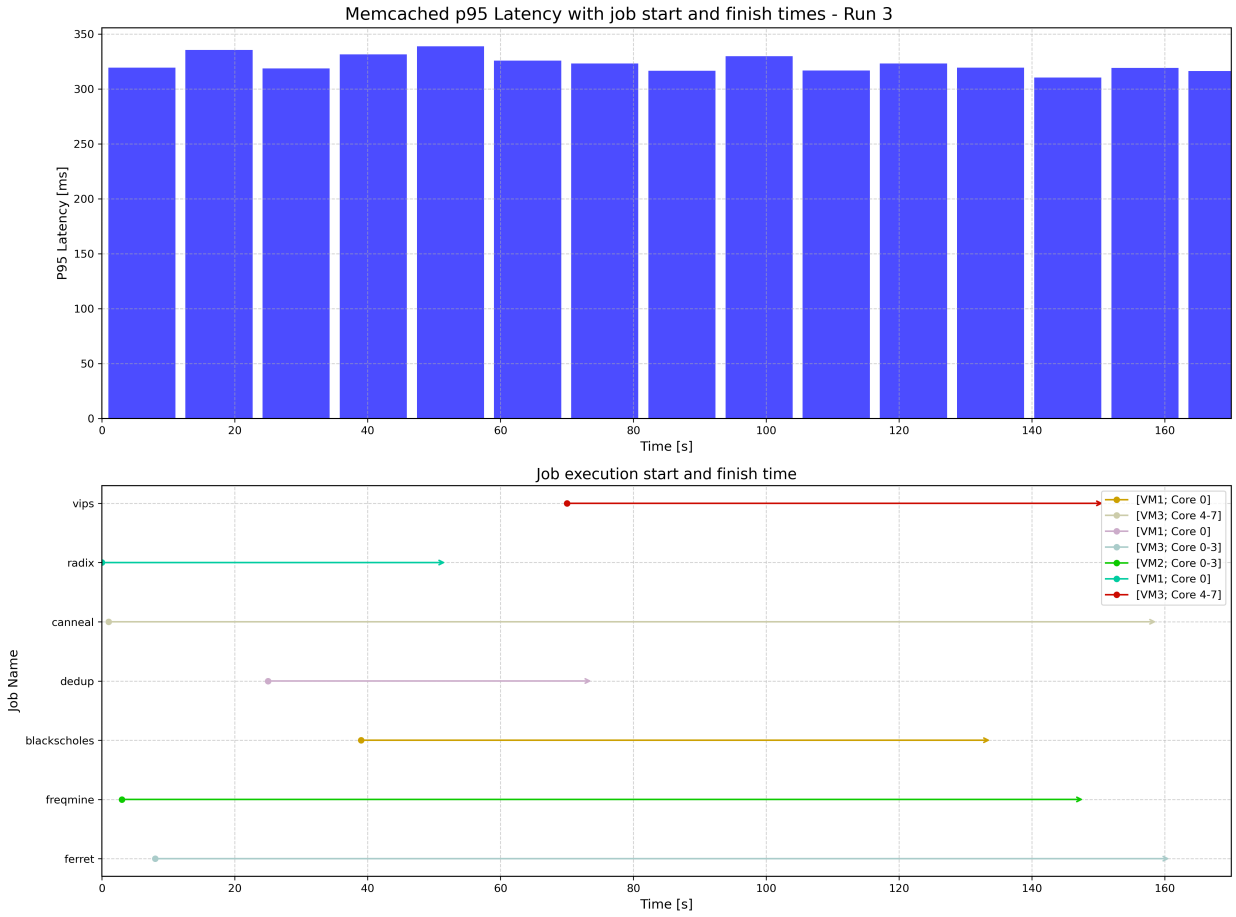

Figure 3: Evolution of p95 latency over the lifetime of parsec jobs - Run 3

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

  Answer: We use a greedy approach to define our policy. We look at the single core run time of each job obtained from part 2b and then assign cores to minimize the run time taking into account mainly the speedup gain from parallelism and a bit of the resource requirements of each job.

  - blackscholes: *duration of 120s* - We schedule the job on core 0 of node-a, i.e. we collocate it with memcached. It has only limited benefits from parallelism, and it has little sensitivity to all interference tyes. Hence it should not compete with memcached for resources.
  - canneal: *duration of 250s* - We schedule it on node-c, cores 4-7. Although it is less parallelizable compared to other jobs, the base duration of the job is medium. This means, the albeit smaller speedup gains are still important. Furthermore, it has in general not a high interference sensitivity besides last-level cache, making it a good candidate to collocate with ferret, which is quite sensitive to almost all interference types.
  - dedup: *duration of 20s* - Although this job is susceptible to interference, it is very short in execution time. Hence its impact will be limited and we schedule it on node-a, core 0.
  - ferret: *duration of 320s* - Has a similar profile to freqmine with a slower execution duration. Hence we run it on node-c, cores 0-3. This should enable the job to benefit from parallelism. Because no other job is collocated with ferret on the same cores, interference should be minimal.
  - freqmine: *duration of 500s* - Part 2 shows that it receives significant speedup from parallelism. Furthermore, the previous measurements show its sensitivity to CPU interference. Hence, we put it on node-b with all four cores assigned as it should benefit from both parallelism and the node's fast CPU.
  - radix: *duration of 56s* - We schedule it on node-a, core 0 with blackscholes and dedup. It has small resource requirements and is a very short-lived job. Altough this job can be parallelized well, there are few incentives to do so because its base duration is very short.
  - vips: *duration of 100s* - We schedule it on node-c, cores 4-7 as this job benefits strongly from parallelism. It is collocated with canneal which should not be a problem as both jobs do not expose high resource requirements.

- Which jobs run concurrently / are colocated? Why?

  Answer: Vips and canneal run concurrently on cores 4-7 of node-c. The reason is that they both benefit strongly from parallelism, and moreover, canneal has a good interference profile. Hence, the two jobs should not compete excessively for the same resources. Furthermore, for the total duration we consider it better to have two medium-duration jobs interfere with each other as their total duration is similar to the one of a big job.
  Ferret is located on the same CPU but on cores 0-3. While it also benefits from parallelism, it is susceptible to CPU interference and L1 instruction cache interference, which would lead to longer execution times when collocated with other jobs on the same cores. Now, there might still be interference on the last level cache and memory bandwidth but the effects should be minimal.
  We collocate blackscholes, radix and dedup together on core 0 of node-a. Although radix and dedup are susceptible to some interference types, they are both very short -

hence, the total duration should stay short even if they affect one another. Blackscholes is much less susceptible to interference and does not expose large speedup gains when being parallelized.

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

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

  Why: **We run all jobs at the same time.** The reason is simple: the bottleneck/near-bottleneck job is freqmine which is consistently the longest (or close second longest) job. Because freqmine is already running on the four fastest cores without collocation, there is no clear way to improve the total running time. Although one might improve the running time of e.g. blackscholes, radix or dedup by not having dedup and radix run concurrently on the same core, we do not expect great improvement in total running time.

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

  Answer:

  - blackscholes: One, because it can benefit less from parallelization, which is worth spent on longer running jobs that can make better use of it.
  - canneal: Four, because the single-thread duration is long and the job benefits from parallelization.
  - dedup: One, because there is no point in improving the run time of this job as it is not a bottleneck with its short single-threaded execution time.
  - ferret: Four, because the single-thread duration of this job is long and it benefits significantly from parallelism.
  - freqmine: Four, for the same reason as ferret. It gains great speedup when multiple threads are used.
  - radix: One, although it benefits from parallelism, the job itself does not take much time.
  - vips: Four, as it is a long-running job and can benefit from parallelism.

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

  Answer: We modified the parsec yaml files to use `taskset` in order to assign specific cores to our jobs. The same command was used to set core 1 to memcached. Apart from that we did other modifications in the command (or its arguments) options of the yaml files to ensure the appropriate level of parallelism for the parsec jobs. With the node selector attribute we allocated the jobs to their corresponding VM. We tried additional features like Kubernetes requests/limits. However, there was no significant improvement when trying to do more fine-grained resource allocation than the current static and unordered policy.

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

  Answer: We chose to optimize greedily the time of the longest jobs. The single-threaded benchmark of part 2 were used. To do this, we first thought about how many threads and cores to assign to each job. We then distributed the jobs on the machines (and the specific cores) by primarily keeping their interference profiles in mind. Additionally, we had a look at the expected run time of each machine and chose to collocate for example two medium-duration jobs together. This strategy quickly lead to a solution where all

VMs end at roughly the same time. Since the bottleneck for most measurements was node-b with freqmine (which has no collocations and the node has fast CPU), we decided not to try further manual optimizations.

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:

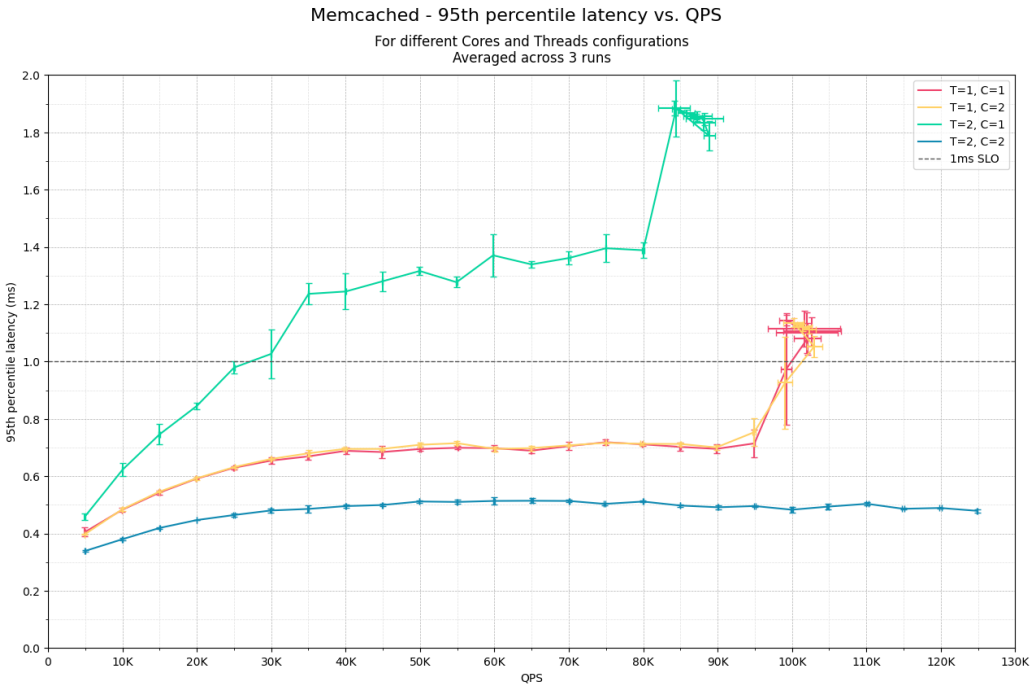

Figure 4: Memcached performance for different Cores and Threads configurations

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 can observe that assigning a single thread to memcached gives similar performance results regardless of the number of cores. However with two threads, memcached performs clearly worse when it is only run on one core. Adding a second core yields stable memcached performance over the whole QPS range.

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 which managed to support 125K QPS is two cores and two threads.

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: Having observed that a single thread cannot handle any load above 110K QPS, the answer is 2 threads. A single core is able to handle the workload until around 30K QPS, where a second core would have to used.

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:
Figure 5

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 jobs as soon as possible without violating the 1ms 95th percentile latency for memcached.

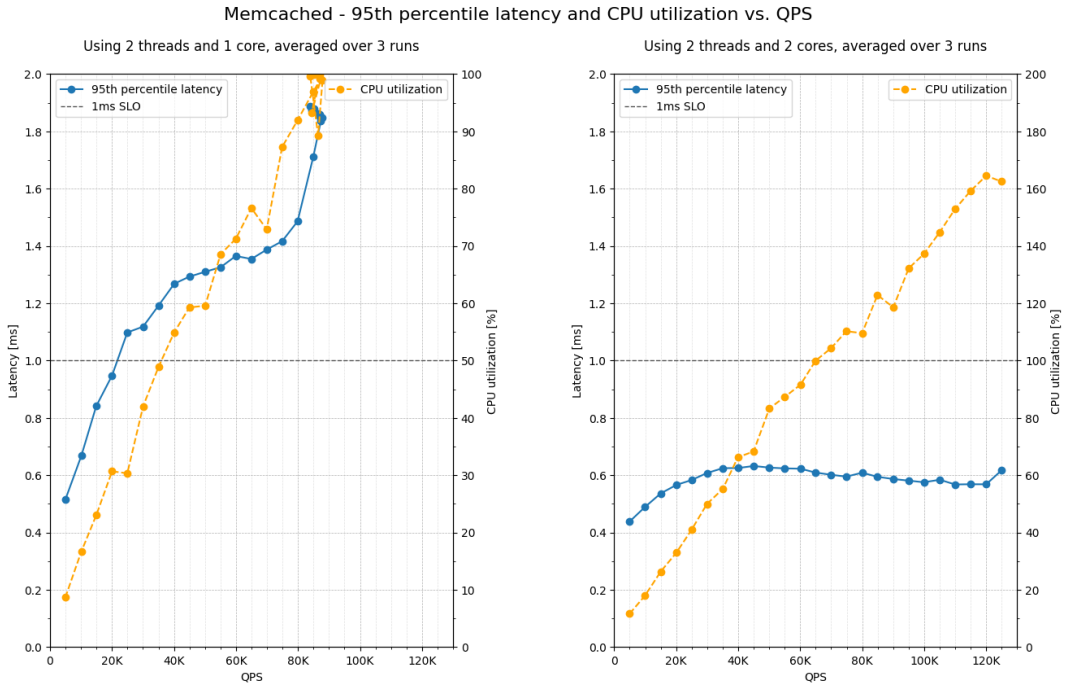

Figure 5: Memcached performance and CPU utilization using 2 threads on 1 and 2 cores.

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: Our goal was to create a simple, yet powerful scheduler that is able to run all jobs in a configurable manner by minimizing interference between the jobs as well as between memcached and the jobs. We decided to constantly allocate two cores to memcached (core 0 and 1), but to schedule low-interfering jobs on core 1 whenever the load on memcached is low. Additionally, we scheduled resource-intensive jobs that are hard to parallelize first. If one job finishes, it forwards the cores on which it ran to the predefined next job(s).

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

  Answer: Memcached is always running on cores 0 and 1. In order to serve the requests within the required SLO at all QPS rates, running memcached on 2 cores with 2 threads is necessary and sufficient.

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

  Answer:

- blackscholes: Based on its interference profile, we expected this workload to be a good fit to collocate with memcached. This is why blackscholes runs on core 1 together with memcached. However, only if the load is low (according to our metric) the container is run.
- canneal: With the same policy as for blackscholes we also run this workload on core 1. For all of our measurements, this workload was running until the end (where all other jobs already finished). It then used cores 2 and 3 to finish up.
- dedup: Due to its short execution time and bad parallel behavior, this job was run on its own on core 2. Another reason is its high resource requirement.
- ferret: Ferret was assigned one core 3 at the beginning. After dedup finished, it additionally got core 2 to run on.
- freqmine: Freqmine was assigned two cores (2 and 3). After canneal finished, core 1 would have been assigned to it. However, we never reached this situation.
- radix: Due to its good speedup when run on many threads, we assigned the two cores (2 and 3) to radix. Our initial thought was to run it on three cores with 3 threads however we didn't manage to get the container running with an odd amount of threads.
- vips: With the same reasoning as with freqmine, we run this job on two cores. If freqmine finished early on core 1, we would have dynamically assigned core 1 as well.

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

  Answer:
  - blackscholes: One thread as we wanted to run it in parallel with many jobs (memcached, dedup, ferret) and because of its low speedup increase when run with many threads.
  - canneal: Two threads as we found canneal to be running until the very end. It can hence exploit the capacity of the remaining cores.
  - dedup: Since we can observe a bad speedup by using multiple threads and its short execution time, we set this to one.
  - ferret: Ferret exposes great speedup by using two threads and has rather high resource (especially CPU) requirements. We hence chose to run it on two cores by using two software threads.
  - freqmine: Freqmine is a heavy workload as it exposes high CPU usage and a long running time. Since it can make use of parallelization we decided to run it on 3 threads.
  - radix: Radix is a short-running job that can easily be parallelized. However, we were not able to start up a container running the radix workload on 3 threads. This is why we chose to run it on just 2 threads.
  - vips: Vips is also one of the best jobs when it comes to parallelization. We chose to run it on 3 threads because of this reason.

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

  Answer: From parts 1 and 2 we concluded that blackscholes, canneal and radix should be good candidates to be coallocated with memcached. Additionally, blackscholes and canneal expose a low speedup curve when running on multiple threads. We hence decided to run both on the same core (core 1) as memcached as long as the CPU usage of memcached is below a certain threshold (40% in our case). The other jobs are more CPU

intensive (as observed in measurements of part 2.1). This is why we were conservative when it came to other collocations and decided not to implement this in our policy.

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

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

  Why: We put the jobs that are not easy to parallelize first. The reason being that at a later point in time when we only have a small amount of jobs remaining, we can make use of several cores per job. This will hence increase efficiency compared to a situation where a difficult-to-parallelize job would be run on several cores. Our scheduler started with blackscholes (core 1), dedup (core 2) and ferret (core 3). Then ferret takes over core 2 (from dedup) before freqmine, vips and radix are run in this order. Canneal is started after blackscholes on core 1 due to its longer execution time and its worse interference profile.

- How does your policy differ from the policy in Part 3? Why?

  Answer: Part 3 uses a heterogeneous cluster with several VMs. The SLO is also less restrictive than the one in part 4. This is why part 3 is able to run memcached on just one core with one thread. CPU-intensive tasks are run on other VMs resulting in less interference with memcached and faster execution times. In part 4 we just have 4 cores available and due to the interference-profiles of the jobs we could not run them concurrently without having severe interference. So the main difference is the sequential execution in part 4 versus the parallel execution in part 3. The controller in part 4 also has to dynamically adjust core affinity or container execution to exploit the QPS fluctuations whereas part 3 has a constant QPS and can therefore use static scheduling.

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

  Answer: From `psutil.Process` we used the `cpu_percent()` method to capture CPU usage of memcached. By "handing over" cores from one job to another we used `cpu_set()` from the `docker-py` library. In order to run memcached on cores 0 and 1 we used the `taskset` command. Pausing/Unpausing of the containers was used whenever memcached showed high/low CPU usage and the container ran on the same core as memcached.

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

  Answer: From part 4.1 we knew that memcached needs to run on 2 cores with 2 threads in order to achieve the SLO at maximum QPS. We hence reserved the first 2 cores for memcached and ran it on 2 software threads. The second design decision was on how many threads to run the different jobs. We took the execution time and the parallel speedup from part 2 into account. Next, we decided on which cores to possibly run the benchmarks. For the resource-intensive tasks (dedup, ferret, freqmine, vips) we decided to run them sequentially and on different cores than memcached to mitigate interference. Decreasing thread amount was used to define the sequential order of the jobs for better efficiency towards the end of execution. When deciding on which granularity to check memcached's CPU usage and whether the container of the jobs were finished, we came to the conclusion to do this every 0.25 seconds. This allows fast reaction to state changes while having little overhead.

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: We have a SLO violation ratio of 0%. For no datapoint the SLO is violated.

job name	mean time [s]	std [s]
blackscholes	102.00	1.37
canneal	143.47	4.58
dedup	40.24	2.37
ferret	208.36	0.42
freqmine	395.40	7.01
radix	23.07	0.71
vips	52.80	0.51
total time	755.12	13.39

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:
Figures 6, 7, 8, 9, 10 and 11
In order to show how our policy works and observe the correlation between stopping jobs running on core 1 whenever the load on memcached is high, we always included another plot instead of annotating the other two plots. This (additional) plot shows which job is running on which core at a given time.

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:

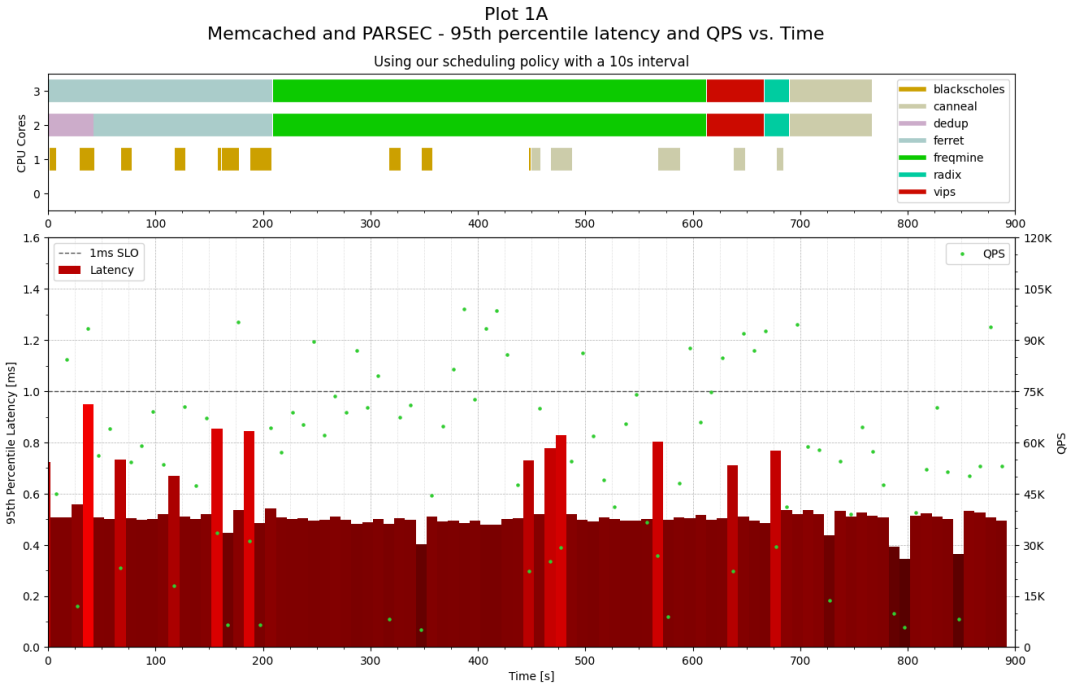

Figure 6: Parsec jobs core allocation and memcached performance using our scheduling policy with a 10s QPS interval - Run 1

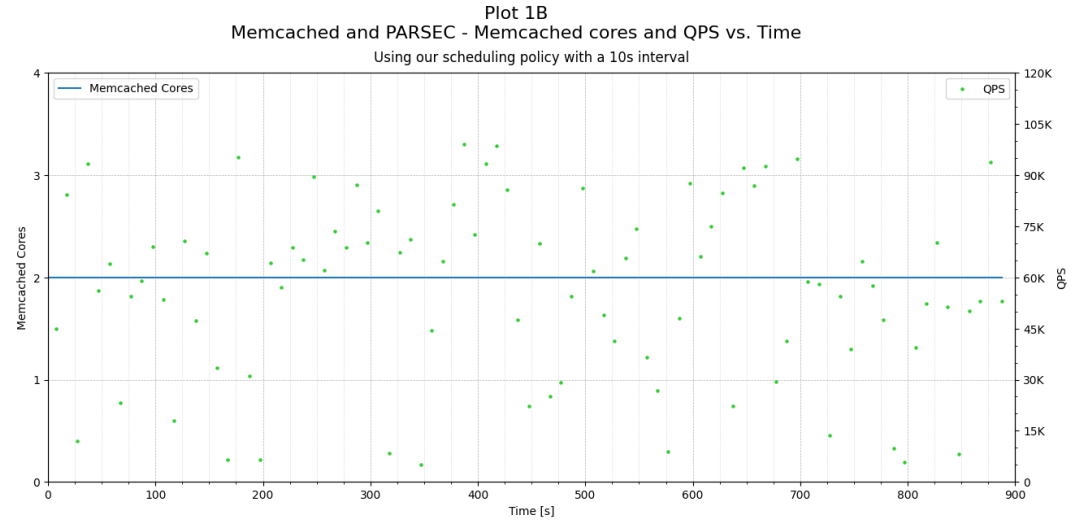

Figure 7: Memcached core allocation using our scheduling policy with a 10s QPS interval - Run 1

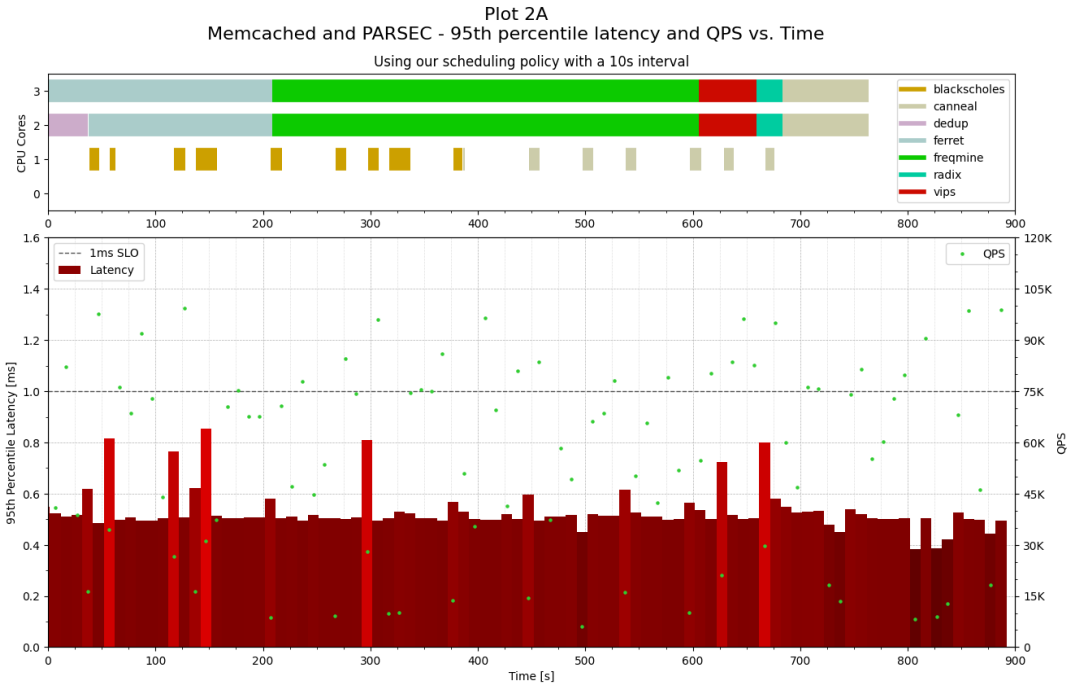

Figure 8: Parsec jobs core allocation and memcached performance using our scheduling policy with a 10s QPS interval - Run 2

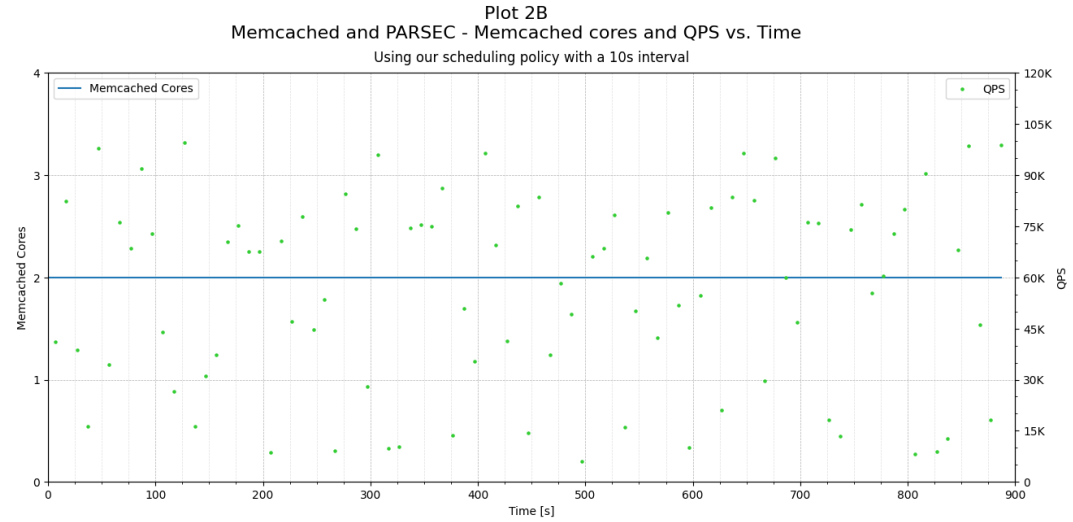

Figure 9: Memcached core allocation using our scheduling policy with a 10s QPS interval - Run 2

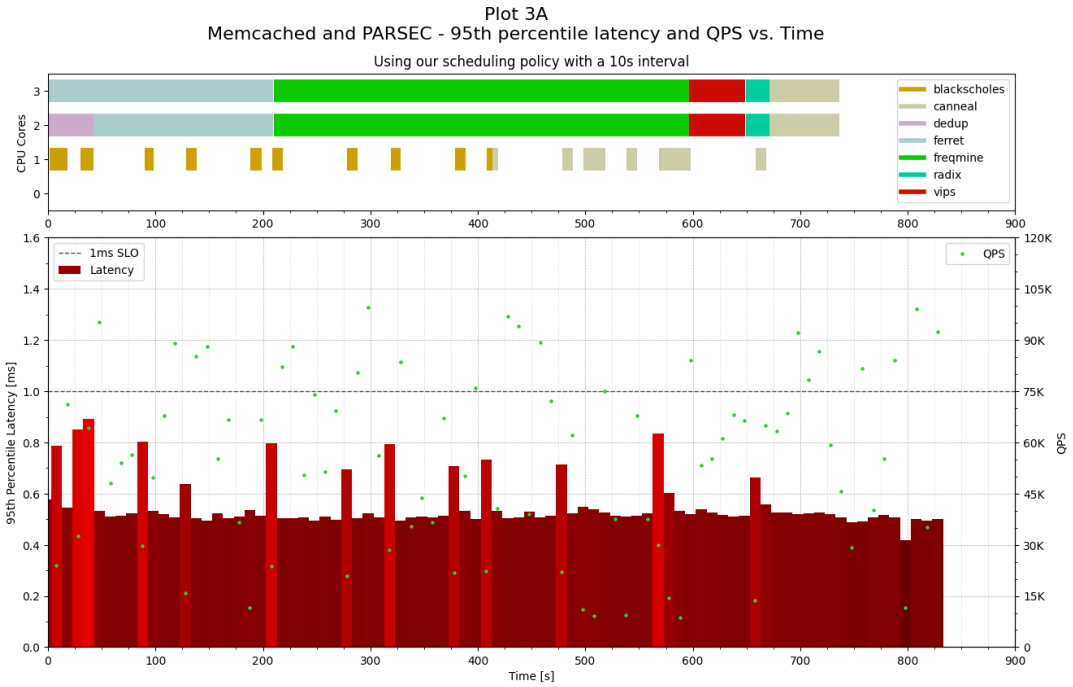

Figure 10: Parsec jobs core allocation and memcached performance using our scheduling policy with a 10s QPS interval - Run 3

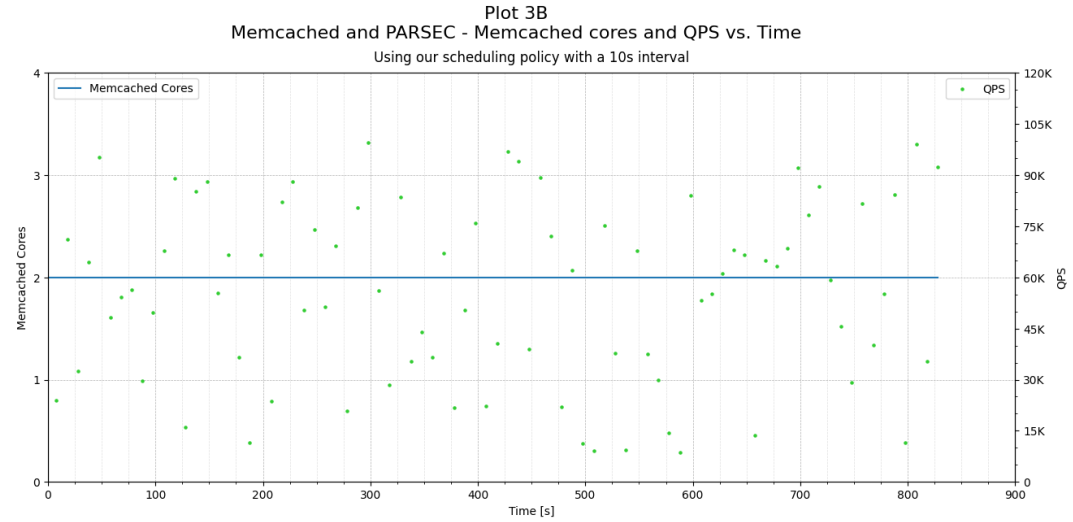

Figure 11: Memcached core allocation using our scheduling policy with a 10s QPS interval - Run 3

```
$ ./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: We achieve a very similar performance compared to task 4.3. The total time is surprisingly smaller (by around 20 seconds) than in the 10 seconds interval case. The individual jobs require almost the same times.

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: We still have no SLO violation when our controller is run with this smaller `qps_interval`.

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: We found the minimal `qps_interval` to be 1 second. With this value we get an average SLO violation of about 2.36%. We also tried to run with a `qps_interval` of 0.5 seconds but this resulted in a higher SLO violation ratio.

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

Answer: Our controller waits for 0.25 seconds after each working step. This allows fast reaction to QPS changes while requiring not much CPU resources on its behalf. Another reason that affects the minimal `qps_interval` is the working step itself. The controller checks the CPU usage of memcached and updates/checks the remaining jobs, which also requires time. If the controller can not react fast enough to higher QPS rates (i.e by freeing resources on core 1), it will adapt slower and hence tail-latency increases.

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	103.52	1.10
canneal	165.51	1.37
dedup	43.80	4.85
ferret	209.70	3.00
freqmine	390.48	1.38
radix	22.83	0.54
vips	50.05	1.64
total time	724.23	4.93

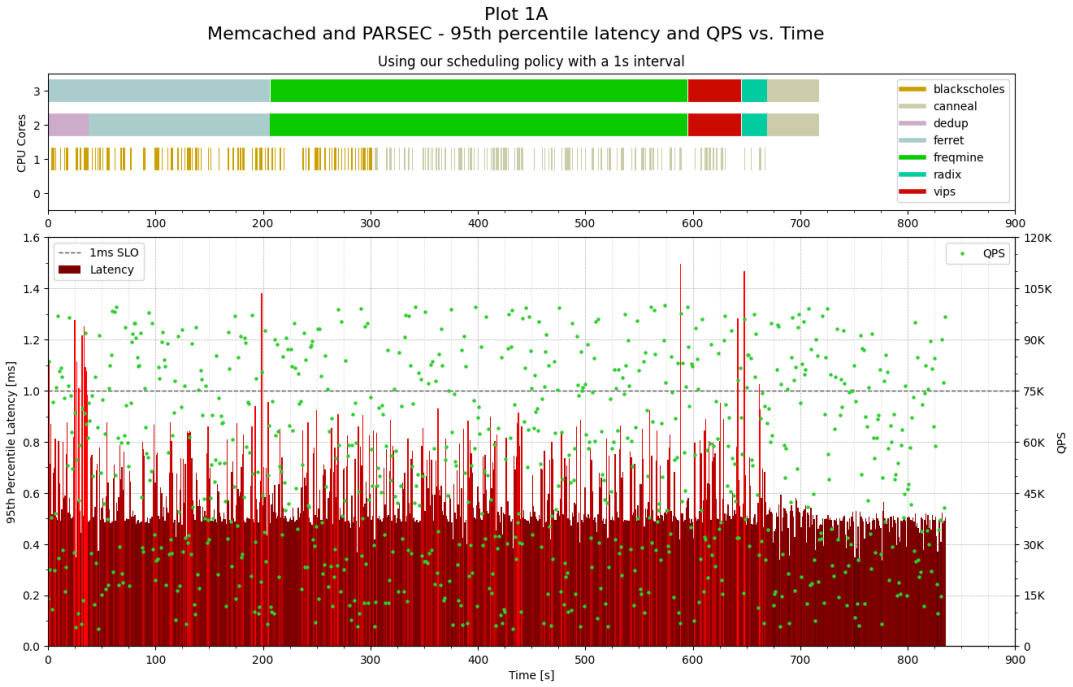

Figure 12: PARSEC jobs cores allocation and Memcached performance using our scheduling policy with a 1s interval - Run 1

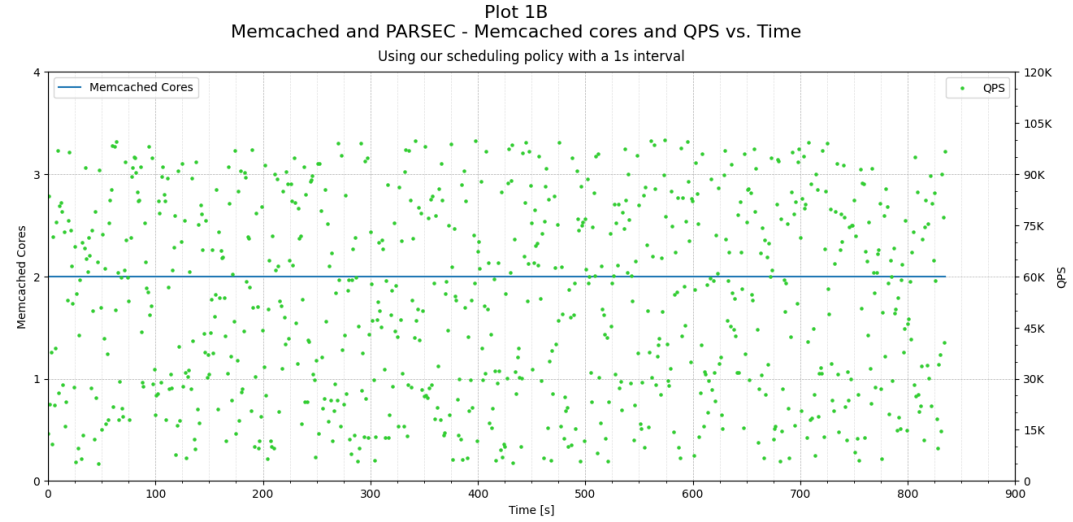

Figure 13: Memcached core allocation using our scheduling policy with a 1s interval - Run 1

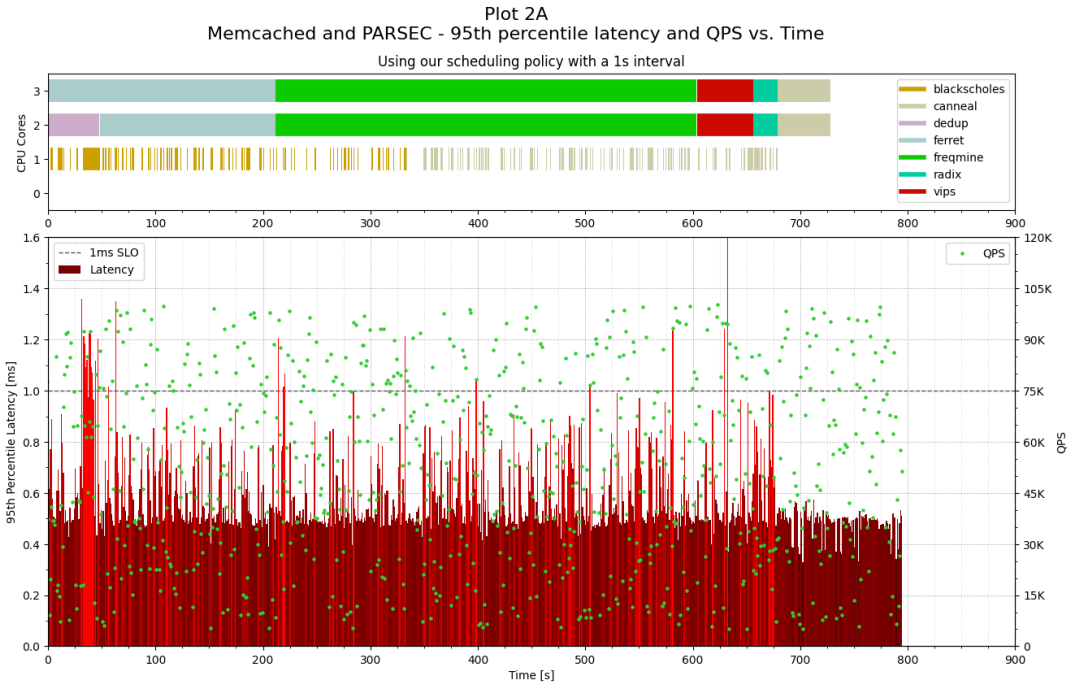

Figure 14: PARSEC jobs cores allocation and Memcached performance using our scheduling policy with a 1s interval - Run 2

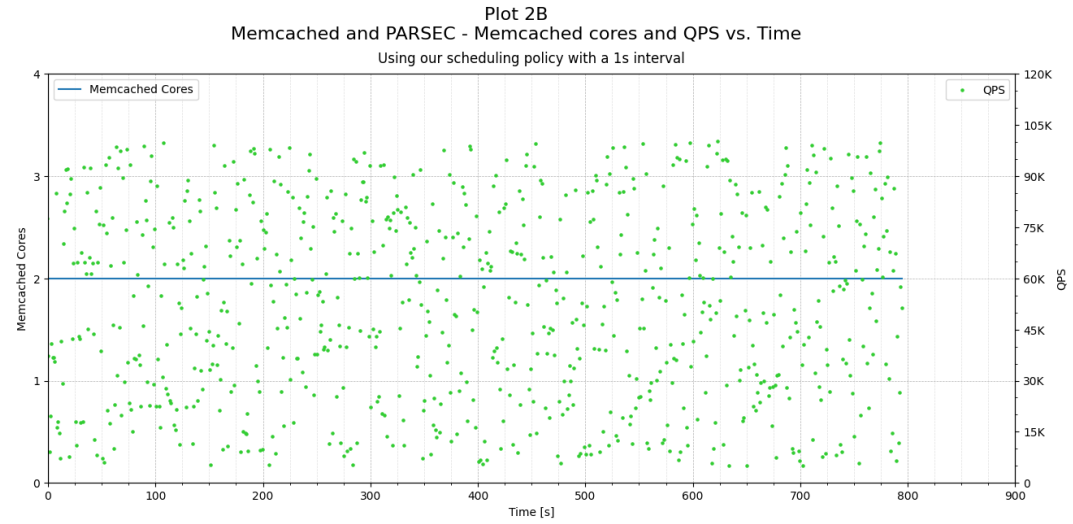

Figure 15: Memcached core allocation using our scheduling policy with a 1s interval - Run 2

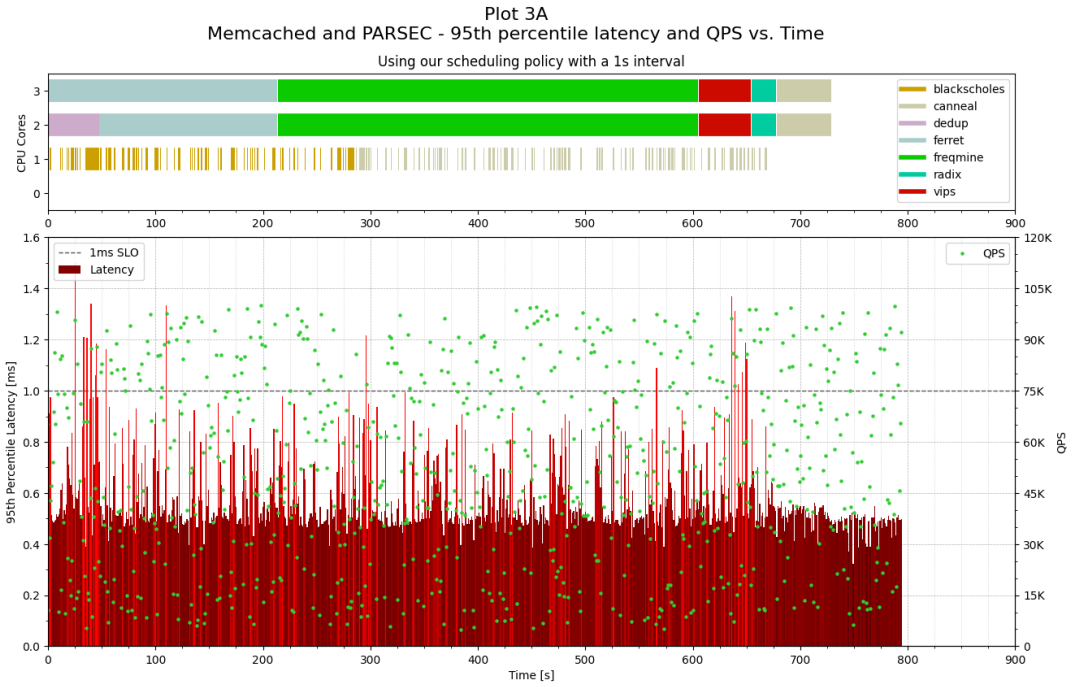

Figure 16: PARSEC jobs cores allocation and Memcached performance using our scheduling policy with a 1s interval - Run 3

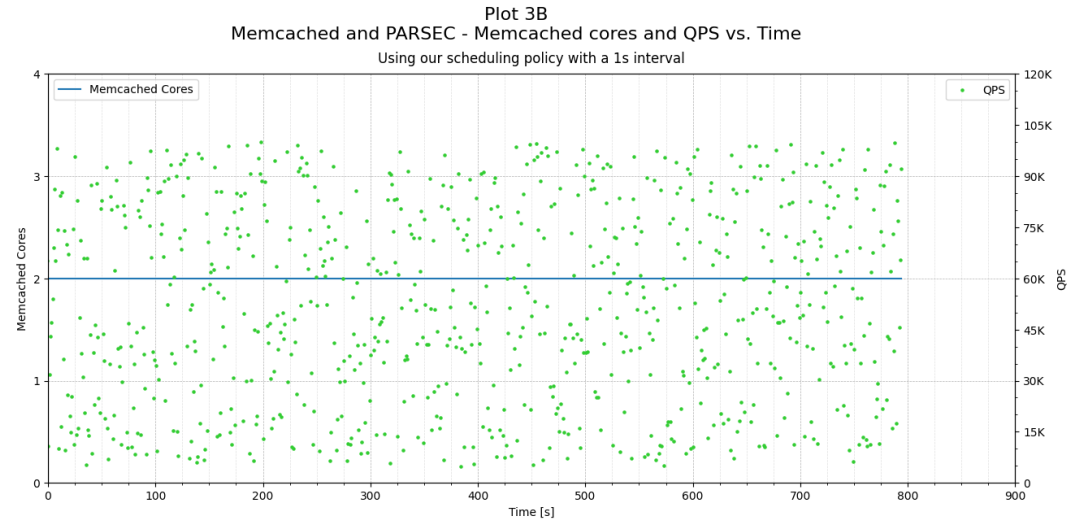

Figure 17: Memcached core allocation using our scheduling policy with a 1s interval - Run 3