Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Cloud Computing Architecture

Semester project report

Group 032
Angelis Marios - 22-953-186
Xypolitos Georgios - 22-943-906
Tsolakis Georgios - 19-953-678

Systems Group
Department of Computer Science
ETH Zurich
April 6, 2023

Part 1 [25 points]

Using the instructions provided in the project description, run memcached alone (i.e., no interference), and with each iBench source of interference (cpu, l1d, l1i, l2, llc, membw). For Part 1, you must use the following `mcperf` command, which varies the target QPS from 30000 to 110000 in increments of 5000 (and has a warm-up time of 2 seconds with the addition of `-w 2`):

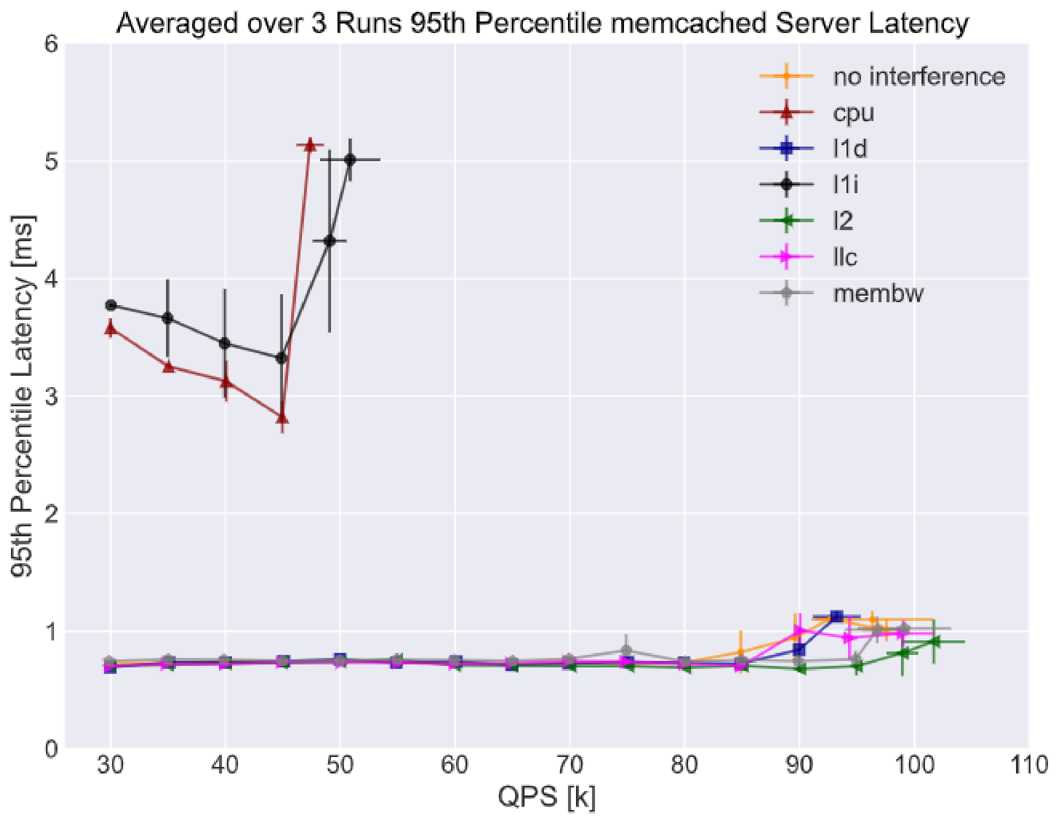
```
$ ./mcperf -s MEMCACHED_IP --loadonly
$ ./mcperf -s MEMCACHED_IP -a INTERNAL_AGENT_IP  \
           --noload -T 16 -C 4 -D 4 -Q 1000 -c 4 -w 2 -t 5 \
           --scan 30000:110000:5000
```

Repeat the run for each of the 7 configurations (without interference, and the 6 interference types) **at least three times** (three should be sufficient in this case), and collect the performance measurements (i.e., the `client-measure` VM output). Reminder: after you have collected all the measurements, make sure you delete your cluster. Otherwise, you will easily use up the cloud credits. See the project description for instructions how.

(a) **[10 points]** Plot a single line graph with the following stipulations:

- Queries per second (QPS) on the x-axis (the x-axis should range from 0 to 110K). (note: the actual achieved QPS, not the target QPS)
- 95th percentile latency on the y-axis (the y-axis should range from 0 to 8 ms).
- Label your axes.
- 7 lines, one for each configuration. Add a legend.
- State how many runs you averaged across and include error bars at each point in both dimensions.
- The readability of your plot will be part of your grade.

Solution:
The interference introduces resource contention and the target QPS cannot be achieved after approximately 50k QPS for l1i and cpu, after 95k QPS for l1d, and after 100k QPS for all the others. This results in multiple overlapping points and poor plot readability. To address this problem we averaged the overlapping points into one and took all of them into account to create the horizontal error bar, expressing the range of those values. Lastly in order to enhance the readability of our plot even more we decided to limit our y-axis to the value of 6.



(b) **[6 points]** How is the tail latency and saturation point (the "knee in the curve") of memcached affected by each type of interference? Why? Briefly describe your hypothesis.

Solution:

First of all, we observe that the 95th percentile tail latency of `memcached` during the **l1d, l2, llc** and **membw** interference is almost identical to the tail latency without any interference. The saturation point for these interferences seems to be around 90-95k QPS.
For the **cpu** and **l1i**, even for a small number (30k) of QPS, the 95th percentile response time is about 4 times higher than the other types of interference and by 45k QPS, the saturation point is reached. After the saturation point, response times increase even further up to 5 times the baseline response time (no interference). The service is not able to handle more than 50k QPS when there is cpu or l1i interference. This suggests that these types of interference are the most critical and impact latency. This correlation is consistent with the interactive law between response time and throughput introduced in the lecture. The

fact that **l1i** and **cpu** interfere with the benchmark can be explained in two ways. On one hand, **cpu** is used to handle everything, including the actual run of the interference and the benchmark and therefore is naturally overloaded faster. On the other hand, CPU is heavily relying on instructions in the L1 instruction cache, which have to constantly change between the benchmark, the interference and other scheduling and overhead activity of the machine, therefore resulting in conflict causing the response time to be higher.
For the other types of interference, we can assume that for these numbers of QPS, although it seems we are reaching the saturation point (knee-in-the-curve) at about 90k, we still haven't hit the overload region, although we're pretty close, maybe even in a shallow region, which still hasn't managed to crash the system. This comes from the theoretical knowledge that once we hit the overload state, the response time should increase rapidly, as we can see in the CPU and L1i interference lines.

(c) [**2 points**] Explain the use of the `taskset` command in the container commands for memcached and iBench in the provided scripts. Why do we run some of the iBench benchmarks on the same core as memcached and others on a different core?

Solution:
The `taskset` command is used to set the CPU affinity of a process. In order for the interference to actually interfere with the components we want to get correct measurements, we have to be sure that the interference and the benchmark are both running on the same core. This holds for the CPU, L1d, L1i and L2 benchmarks, for which all resources are distinct for each core. However, LLC and membw interferences refer to resources that are shared among different cores. What that means is that for them to interfere only with the intended levels of memory hierarchy, namely LLC and memory, we have to place the corresponding benchmarks on different cores.

(d) [**2 points**] Assuming a service level objective (SLO) for memcached of up to 1.5 ms 95th percentile latency at 65K QPS, which iBench source of interference can safely be collocated with memcached without violating this SLO? Briefly explain your reasoning.

Solution:
All of the interferences except CPU and L1i can be safely collocated with `memcached`, without violating the SLO. This is true, because for the current QPS tests, we can see from the measurements, that only CPU and L1i benchmarks reach a response time of more than 1.5 ms.

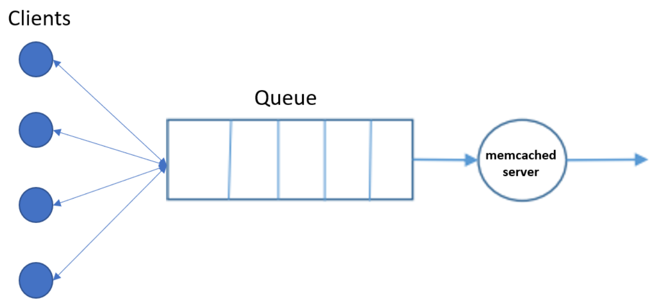
(e) [**5 points**] In the lectures you have seen queueing theory. Is the project experiment above an open system or a closed system? What is the number of clients in the system? Sketch a diagram of the queueing system and provide an expression for the average response time. Explain each term in the response time expression.

Solution:
The project experiment is an open system. In an open system, the number of potential clients is unlimited and the load is not self-adjusting (the load may keep coming even if the system stops). In our example, mcperf client establishes 4 connections with the server. Thus, in our case, from the server's point of view, the number of clients is 4. However, as we mentioned before, the number of clients connected to a server in an open system can be potentially unlimited, for example, other clients can be connected to the same memcached server concurrently. From Little's Law, in open systems, $R = E[w_q] + E[s]$, where R is the

mean response time, $E[w_q]$ is the mean time of waiting in the queue of the system and $E[s]$ is the time the server needs to process one request. This system can be modeled as an M/M/1 system with a single server, an infinite buffer and a First Come First Served queue (FCFS). Thus, the average response time can be modeled as:

$$R = E[w_q] + E[s] = \frac{1/\mu}{1-\rho} = \frac{1/\mu}{1-(\lambda/\mu)} = \frac{1}{\mu-\lambda}$$

where λ is the arrival rate of requests, μ is the mean service rate and ρ is the utilization of the server. Finally, the diagram of the queueing system is presented in the following graph.



Part 2 [30 points]

1. **Interference behavior [19 points]**

   (a) **[11 points]** Fill in the following table with the normalized execution time of each batch job with each source of interference. The execution time should be normalized to the job's execution time with no interference. Round the normalized execution time to 2 decimal places. Color-code each field in the table as follows: **green** if the normalized execution time is less than or equal to 1.3, **orange** if the normalized execution time is over 1.3 and up to 2, and **red** if the normalized execution time is greater than 2. Briefly summarize in a paragraph the resource interference sensitivity of each batch job.

   Solution:

Workload	`none`	`cpu`	`l1d`	`l1i`	`l2`	`llc`	`memBW`
blackscholes	1.00	1.29	1.17	1.57	1.23	1.48	1.34
canneal	1.00	1.12	1.09	1.22	1.08	1.84	1.26
dedup	1.00	1.62	1.28	2.05	1.19	2.19	1.6
ferret	1.00	2.09	1.03	2.32	1.04	2.5	1.94
freqmine	1.00	1.97	1.0	2.01	1.0	1.8	1.58
radix	1.00	1.07	1.06	1.07	1.05	1.56	1.12
vips	1.00	1.63	1.53	1.89	1.51	1.9	1.66

   In general, all the applications slow down when there is congestion in L1 instruction cache since there are many misses in each application's instruction stream. In this paragraph, we briefly summarize the resource interference sensitivity of each batch job.
   Dedup slows down most when there is congestion in the L1 instruction cache, L3 cache, and memory bandwidth. Thus, we can assume that dedup is memory intensive since the

CPU interference affects it less than memory interference. Blackscholes has a similar behavior to dedup affected most by L1 instruction cache, L3 cache, and memory bandwidth interference. On the other hand, ferret has a performance degradation of 2 when CPU interference is applied. Still, it is also highly affected when there is congestion in the L1 instruction cache, L3 cache, and memory bandwidth. We can notice that ferret is not affected by L1 and L2 data cache interference. This is because its workload fits in neither L1 nor L2 cache and it is stored in the L3 cache. Thus, when there is congestion in the L3 cache, the performance degrades a lot. Since ferret is affected more than any benchmark from the memory bandwidth congestion, we can assume that it is the most resource-intensive application. Also, it is affected more than any benchmark by CPU congestion so it is also computation intensive. Freqmine has similar behavior to ferret. It has a performance degradation of 2 when CPU interference is applied, but it is also highly affected when there is congestion in the L1 instruction cache, L3 cache, and memory bandwidth. Canneal and radix have similar behavior and they are only affected by L3 cache congestion. Last, vips has the same behavior for all types of interference, but it is mostly affected by L3 cache congestion. It is important to mention that the results have been computed as the average of two executions.

(b) [**8 points**] Explain what the interference profile table tells you about the resource requirements for each application. Which jobs (if any) seem like good candidates to collocate with memcached from Part 1, without violating the SLO of 2 ms P95 latency at 40K QPS?

Solution:

From the results we obtained, we can assume that ferret and freqmine are both computation and memory intensive. Also, ferret is resource-intensive since it is affected by memory bandwidth congestion more than any other application. Dedup is memory intensive since the CPU interference affects it less than memory interference. Canneal and radix are only affected by L3 cache congestion, so they are memory-intensive applications and finally, vips is also affected by both CPU and data cache congestion so we can assume that it is both CPU and memory intensive. However, compared to ferret, vips is less intensive in both factors.

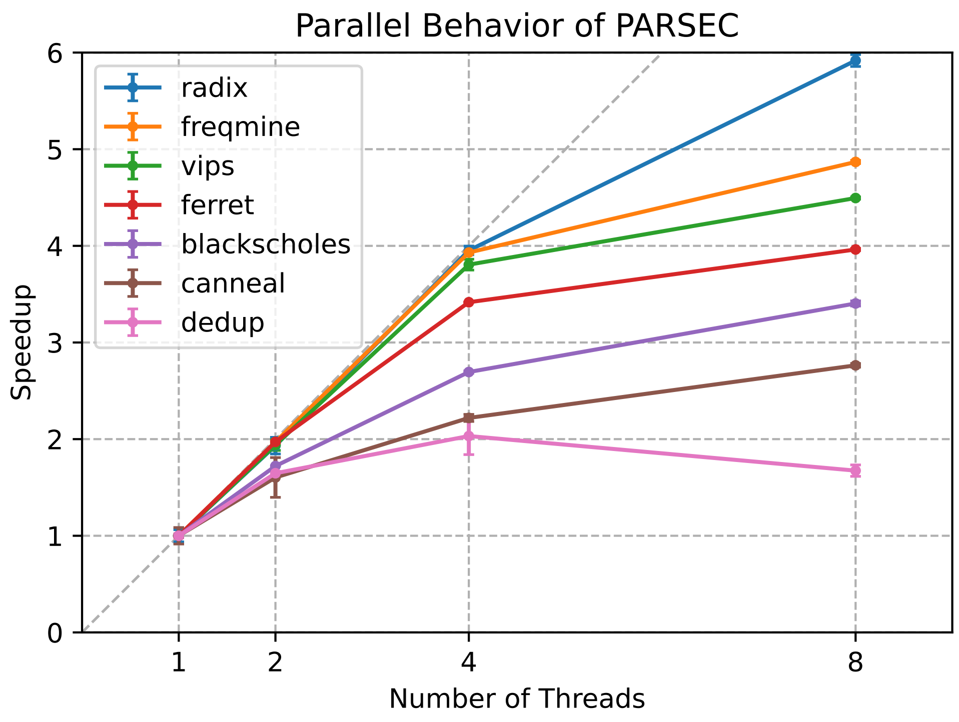
From the first part, we know that memcached performance deteriorates when there is interference in CPU and L1 instruction cache so any application that heavily uses these resources is unacceptable to collocate with memcached. For this reason, ferret, freqmine, dedup, and vips cannot be collocated with memcached because they are CPU intensive. Also, each application suffers from L1 instruction cache congestion, but we consider their use of L1 instruction cache moderate. Thus, bloackscholes is the ideal application to coexist with memcached, but also radix and canneal can also collocate with memcached without causing extra interference.

2. **Parallel behavior [11 points]**

Plot a single line graph with speedup as the y-axis (normalized time to the single thread config, Time_1 / Time_n) vs. number of threads on the x-axis (1, 2, 4 and 8 threads, see the project description for more details). Briefly discuss the scalability of each application: e.g., linear/sub-linear/super-linear. Do the applications gain significant speedup with the increased number of threads? Explain what you consider to be "significant".

Solution:

The interference introduces resource contention and the target QPS cannot be achieved after approximately 50k for l1i and for the cpu, after 90k for l1d and for all the other if we target more than 100k QPS. Thus the points would overlap and



Radix application benefits more than any other application from parallelization. The speedup scales linearly up to 4 threads. From 4 to 8 threads, we have a 50% increase leading to a maximum speedup of 6x. Freqmine and vips have similar behavior scaling linearly up to 4 threads. Then, from 4 to 8 threads, freqmine has a 23% increase reaching a maximum speedup of 4.9x and vips has a 12.5% increase reaching a maximum speedup of 4.5x. Ferret scales linearly up to 2 threads. Then from 2 to 4 threads, the speedup is increased by a factor of 70% reaching a speedup of 3.4x using 4 threads, and from 4 to 8 threads, the speedup is increased by a factor of 18% reaching a maximum speedup of 4x. Blackscholes and canneal scale close to linearly for 2 threads but after that, they are significantly worse than ferret. Finally, dedup loses performance when run with 8 threads compared to 4 threads.

We categorize the measured speedup as follows:

- Linear speedup is identical but no one application manages to achieve this speedup.
- A speedup of more than $\frac{3}{4}*$ threads is considered very advantageous.
- A speedup of $\frac{1}{2}*$ threads is considered still significant.
- Anything less is insignificant.

Thus, radix achieves a very advantageous speedup of 6x. Freqmine, vips and ferret achieve a significant speedup of 4.5x and 4x respectively using 8 threads. Finally, blackscholes, canneal and dedup achieve a significant speedup with 4 threads, but their performance gain from 4 to 8 threads is insignificant, especially for dedup whose performance is worse when run with 8 threads compared to 4 threads. Last, it is important to mention that the results have been computed as the average of two executions.

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.

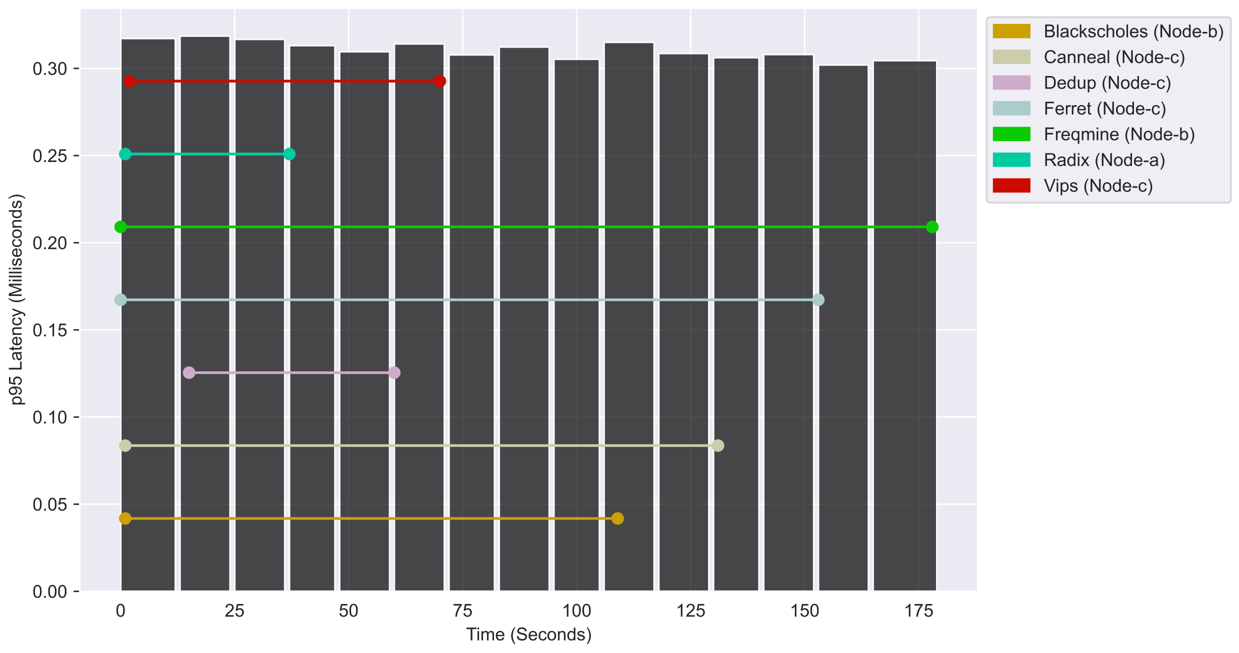
   Answer:
   Create 3 bar plots (one for each run) of `memcached` p95 latency (y-axis) over time (x-axis) with annotations showing when each PARSEC job started and ended, also indicating the machine they are running on. Using the augmented version of mcperf, you get two additional columns in the output: `ts_start` and `ts_end`. Use them to determine the width of each 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	110.66	2.64
canneal	128.66	1.15
dedup	43	2
ferret	152.66	0.57
freqmine	175.66	2.08
radix	36.33	0.57
vips	67.33	1.15
total time	175.66	2.08

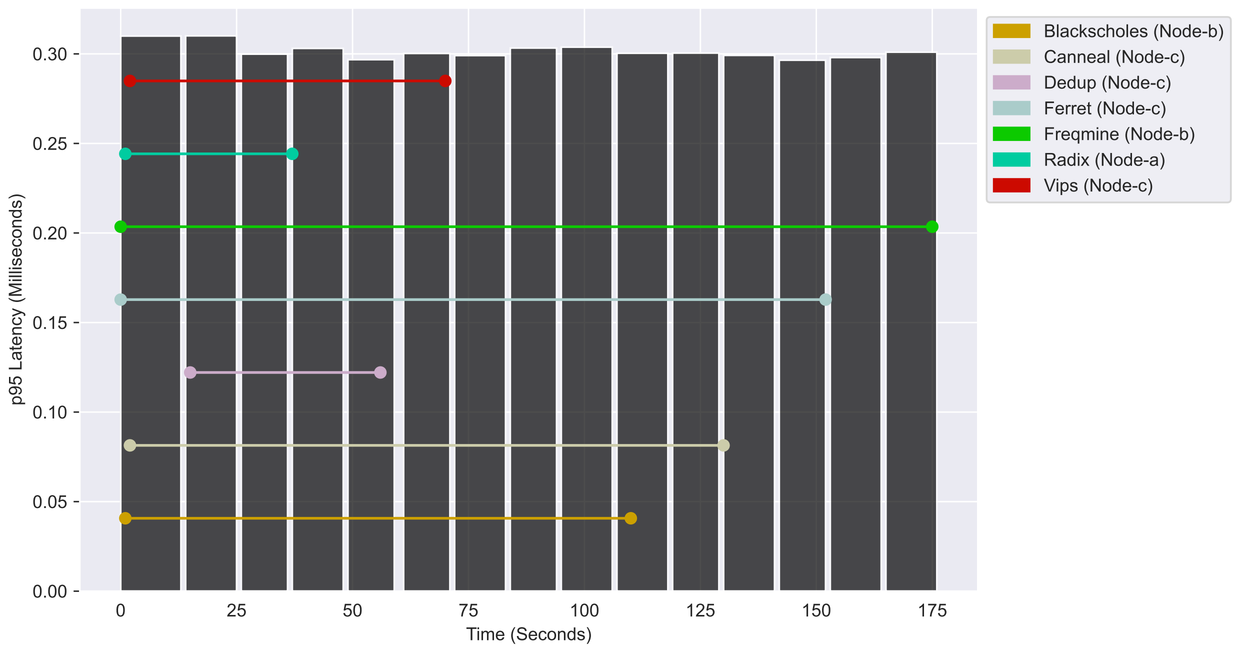
Run	SLO violation ratio (%)
Run 1	0 %
Run 2	0 %
Run 3	0 %

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

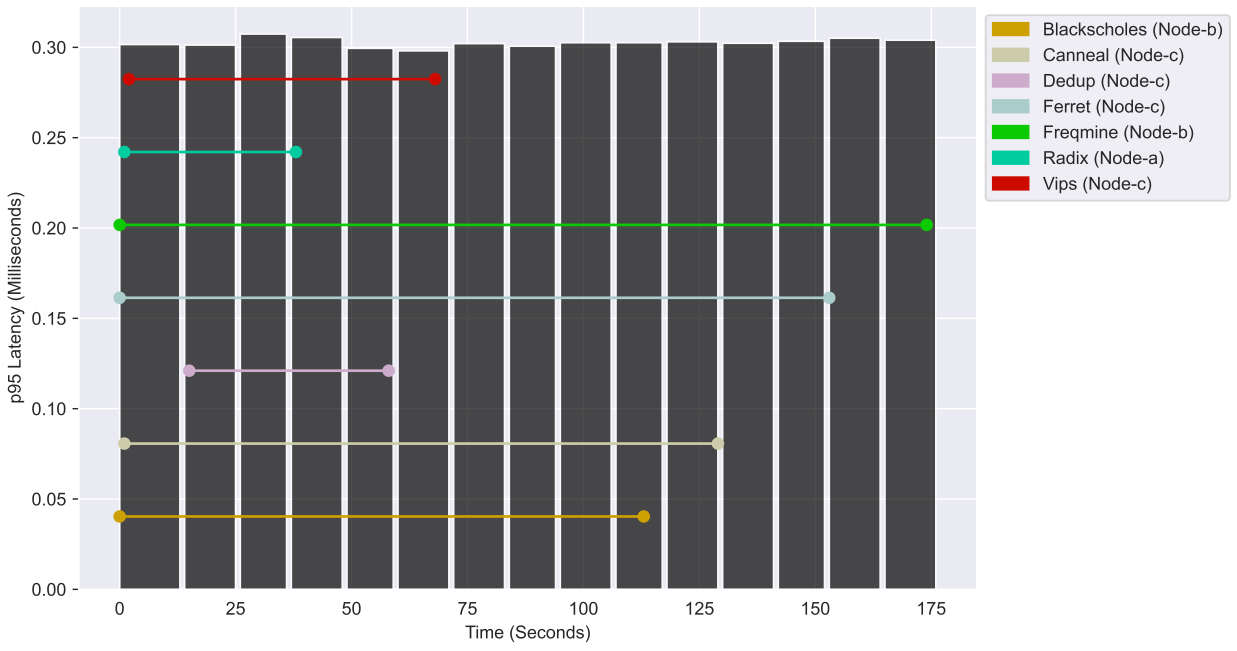
Plot Run 1:



Plot Run 2:



Plot Run 3:



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

- Which node does memcached run on? Why?

  Answer:

  From Part 1, we learned that the `memcached` application is compute-heavy. Thus, the Service Level Objective (SLO) is affected by CPU interference generated from other jobs collocated on the same machine. For this reason, we decided to give `memcached` job its own core to avoid CPU interference. In addition, we observed that `memcached` is not affected by memory interferences, so an optimal scheduling policy could be to place memory-heave applications in the same machine with `memcached` server, trying to keep the SLO below 1 ms. After experimentation using different setups, we concluded that `memcached` job should be placed in **node-a-2-core** machine.

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

  Answer: According to the Google Cloud manual, **node-a-2core** (t2d-standard-2) has 8GB memory and 2 cores, **node-b-4core** (n2d-highcpu-4) has 4GB memory and 4 cores that operate in a higher frequency than the others, and **node-c-8core** (e2-standard-8) has 32GB memory and 8 cores.

  From Part 2, we learned about the impact of CPU and memory interference on the runtime of each PARSEC job. More specifically, we can assign each PARSEC job into one of 3 categories: compute-heavy applications, memory-heavy applications and both compute and memory heavy applications. Thus, `memcached`, ferret and freqmine belong to the compute-heavy category, so they may generate CPU interference to other jobs that are collocated in the same machine. For this reason, we decided to place each one of them in a different machine. Radix, blackscholes and canneal belong to the memory-heavy category, so they may generate memory interference to other jobs that are collocated in the same machine. Finally, vips and dedup are both compute-heavy and memory-heavy applications, so they may create either memory or CPU interference or both of them to other jobs that are collocated in the same machine.

- radix: We placed the radix job on **node-a-2core** since radix is memory-heavy and it will not generate CPU interference to the `memcached` job that runs on the same machine. Upon experimentation, we found that if we place one more memory-heavy job (like blackscholes) on the same machine, the SLO would be violated.
- freqmine: We placed the freqmine job in **node-b-4core** since freqmine is compute-heavy and node-b-4core machine consists of high frequency cores that will lead to performance improvement and a reduction of the total execution time.
- blackscholes: We placed the blackscholes job on **node-b-4core** since blackscholes is memory-heavy and it will not generate CPU interference to the freqmine job that runs on the same machine.
- ferret: We placed ferret job in **node-c-8core** since ferret is compute-heavy and node-c-8core machine has many cores. We could have placed ferret in node-a-2core or node-b-4core, but as we mentioned, since `memcached`, ferret and freqmine are compute-heavy, we placed each one in a different machine.
- vips: Vips is both memory and compute heavy. Therefore, we placed vips job on **node-c-8core**, since through experimentation, we found out that it will not interfere with the ferret job that runs on the same machine.
- canneal: We placed canneal job **in node-c-8core** since node-c-8core has 32 GB of memory and it can sustain more than one memory-heavy jobs like canneal.
- dedup: Dedup is both memory and compute-bound. We placed dedup job in **node-c-8core** since upon experimentation, we found out that it will not interfere with ferret, vips and canneal that coexist in the same machine.

- Which jobs run concurrently / are colocated? Why?
  Answer:
  node-a-2cores: We placed `memcached` and radix in node-a-2core. Radix is a memory-heavy job so it does not generate CPU interference to the memcached job which is compute-heavy. Upon experimentation, we observed that if we place one more memory-heavy job (like blackscholes) in the same machine, the SLO would have been violated.
  node-b-4cores: We placed freqmine and blackscholes in node-b-4-cores. Based on the results we obtained from part2, freqmine is one of the most demanding compute-heavy jobs in terms of CPU. For this reason, since node-b-4core machine consists of high frequency cores, it was the best candidate to place the freqmine job. Then, we decided to place blackscholes in node-b-2cores. Indeed, if we have placed canneal instead of blackscholes together with freqmine, the total execution time would have increased. Also, we cannot place ferret in the same machine with freqmine since both of them are compute-heavy and they will create interference to each other.
  node-c-8cores: We placed ferret, dedup, vips and canneal in node-c-8cores. As we mentioned, we placed `memcached`, ferret and freqmine jobs in 3 different machines since all of them are compute-heavy applications. Also, we placed canneal together with ferret since canneal is memory-heavy and it will not create CPU interference to ferret. Finally, we decided to place dedup and vips in node-c-8cores. Indeed, we observed that if we

have placed vips or dedub in one of the other 2 machines, the total execution time would have increased.

- In which order did you run 7 PARSEC jobs? Why?
  Order (to be sorted): freqmine, ferret, canneal, blackscholes, vips, dedup, radix
  Why:
  We use **kubectl create** command in order to create PARSEC jobs trying to start the most time consuming jobs first. We observed that if we start all jobs at the same time, we achieve the smallest total execution time. It is important to mention that that dedup job always started later than the other jobs. Dedup runs in node-c-8core together with ferret, canneal and vips. We start ferret, canneal and vips before dedup, so dedup container needs more time to be generated since most of the CPU has been already assigned to the other jobs.
- How many threads have you used for each of the 7 PARSEC jobs? Why?
  Answer:
  We noticed that for most of the PARSEC jobs, we achieve much better results if we don't let the OS scheduler to take care of placing the jobs to the CPU cores of each machine. Thus, we pinned the workloads to specific cores directly using the **taskset** command. Also, we set the total number of user level threads that will be used for each job by modifying the **-n parameter** of the args field. The optimal policy regarding the number of threads/cores assigned to each job was chosen based on different experiments we run in the cluster.
  - blackscholes: Blackscholes job runs on node-b-4cores together with freqmine. Since blackscholes is a memory-heavy job, we let it run on any core, while also leaving the default value for the number of threads. Through experimentation, we found out that the total execution time of blackscholes job is minimized if we don't set any specific parameters ourselves.
  - freqmine: Freqmine job runs on node-b-4cores together with blackscholes. As we mentioned, freqmine job is the most demanding job in terms of CPU. Thus, we decided to assign all the cores to freqmine and we also set -n parameter to 4 user level threads. Using this setup, we measured the higher performance regarding the freqmine job.
  - radix: Radix job runs on node-a-2cores together with `memcached`. Since radix is a memory-heavy job, we didn't use neither the taskset command in order to pin the workload to specific core, nor the -n parameter to create numerous user level threads. Upon experimentation, we found out that the total execution time of radix job is minimised if we leave the command as it is.
  - ferret: Ferret job runs on node-c-8cores together with vips, canneal and dedup. As we mentioned before, this job is one of the most demanding jobs in terms of CPU. Thus, we decided to assign cores 0-3 to ferret and we also set -n parameter to 4 user level threads. Using this setup, we measured the highest performance regarding the ferret job.
  - dedup: Dedup job runs on node-c-8cores together with ferret, canneal and vips. Upon experimentation, we decided not to use any specific parameters, since we found out that the total execution time of dedup job is minimized if we don't set any specific parameters ourselves.

  - canneal: Canneal job runs on node-c-8cores together with ferret, dedup and vips. Although canneal is memory-bound, one core was insufficient to provide the best performance. Thus, we decided to assign cores 4-7 to canneal and set -n to 4 user level threads. Using this setup, we measured the highest performance regarding the canneal job as well as the least amount of interference produced to ferret job that runs in the same machine.
  - vips: Vips job runs on node-c-8cores together with ferret, canneal and dedup. Upon experimentation, we decided to assign cores 4-7 to vips and we also set -n parameter to 4 user level threads. Using this setup, vips and canneal use the same cores, however this setup produced the highest performance regarding both vips and canneal jobs as well as the least amount of interference produced to the ferret job that runs on the same machine.

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

  Answer:
  We modified the YAML files of each parsec job and the YAML file of the `memcached` job in order to set the node on which each job will be scheduled using the **nodeSelector** field, as well as set the CPU affinity for each job using the **taskset** command. Also, we set the total number of user level threads that will be used for each job by modifying the -n parameter of the **args** field. In addition, we provide a script called `static_scheduler.sh` that automatically parses each YAML file, makes the appropriate changes and then schedules each job using the **kubectl create** command based on the scheduling policy we described above. Finally, after the completion of all the jobs, the script gathers and analyzes the results and then deletes all the pods and jobs created.

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

  Answer:
  The way we designed and implemented the scheduler was the following: First, we designated heuristically a category to each job from the measurements done in Part 1 and Part 2. This leads to the classification mentioned previously. With the classification in mind, we isolated `memcached` on the 2-core machine, because `memcached` has high CPU demands and is prone to CPU interference. Through experimentation we found out that the 2-core machine is able to service `memcached` without violating the SLO. After deciding on the machine `memcached` would run on, we tried adding memory-heavy jobs on it and check if the added burden would result in violating SLO. We concluded on adding radix alongside `memcached`. On the 4-core machine, which has the highest frequency cores, we chose to run one compute-heavy job (freqmine) and one memory-heavy job (blackscholes), so we utilize the higher CPU frequency, while minimizing the interference introduced by other jobs. Finally, we tried different configurations on the 8 core machine, which resulted in the parameters described previously. Dedup is allowed to run on any core, with the default number of threads. Canneal is memory-heavy, but one CPU wasn't enough, therefore we isolated it on CPUs 4-7, while ferret was isolated on CPUs 0-3. Also, we forced vips to share CPUs with canneal since we concluded that using this setup, vips will not introduce enough interference for the SLO to be violated.

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:

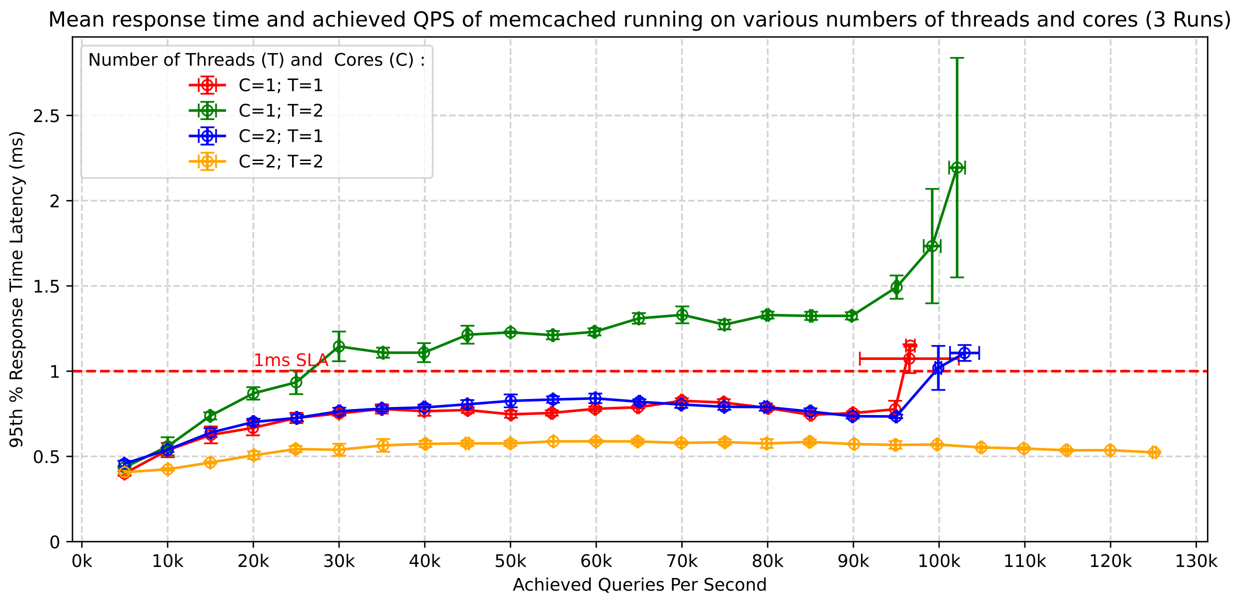
```
$ ./mcperf -s INTERNAL_MEMCACHED_IP --loadonly
$ ./mcperf -s INTERNAL_MEMCACHED_IP -a INTERNAL_AGENT_IP  \
          --noload -T 16 -C 4 -D 4 -Q 1000 -c 4 -t 5 \
          --scan 5000:125000:5000
```

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.

Plot:



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:
The only setup where memcached successfully met the desired QPS and adhered to the 1-millisecond SLA was when T=2 and C=2. None of the other setups were able to achieve either the target QPS or the SLA threshold. From the experimentation, we can observe that in order to handle requests at the higher end of the QPS spectrum, it is necessary to have both two threads and two cores. Also, there is a sharp decrease in performance when both threads have to compete for the same core (C=1, T=2). This is intuitive since both threads are competing with each other for the same set of resources. Additionally, (C=2, T=1) setup demonstrates that in order to fully take advantage of the effect of a multicore setup, multiple threads are needed.

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:
To support load between 5K to 125K QPS and guarantee the 1ms 95th percentile latency SLO, (C=2, T=2) setup should be selected. The above diagram shows that both (C=1, T=1) and (C=2, T=1) setups violate the SLO and do not manage to reach the actual QPS target after 90K QPS. Also, (C=1, T=2) setup violates SLO after 25K QPS, so it should not be selected for the aforementioned goal.

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?

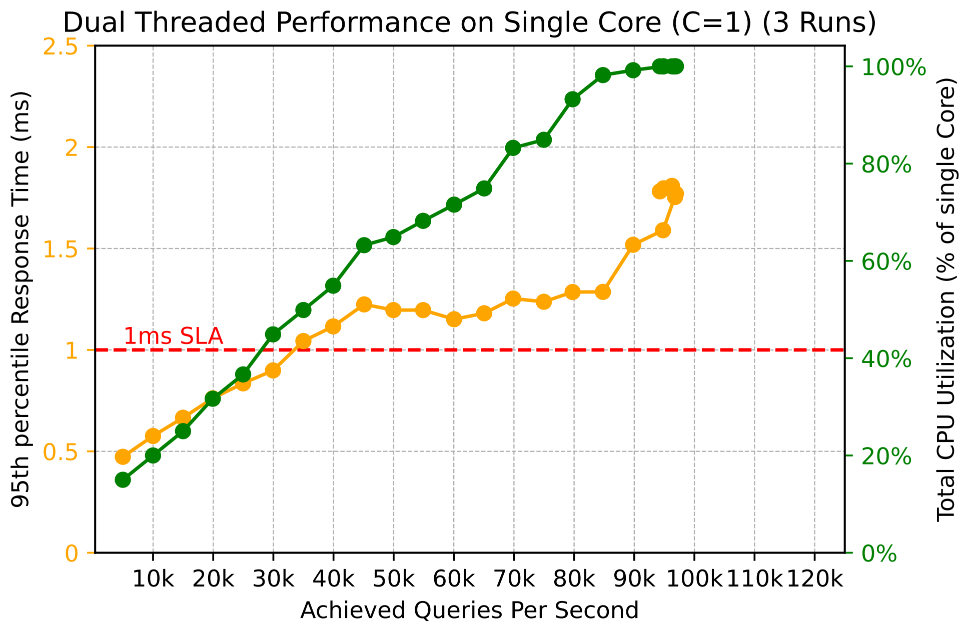
Answer:
In order to guarantee the 1 ms 95th percentile latency SLO while the load varies between 5K to 125K QPS, memcached should use 2 threads (T=2). Then, we can change the number of cores assigned to memcached process dynamically. For example, if the load is smaller than 25K QPS, we can use (C=1, T=2) setup and if the load is larger than this value, we can use (C=2, T=2) setup. If we had chosen 1 thread for the memcached service, none of the available setups would be able to reach the desired goals.

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.

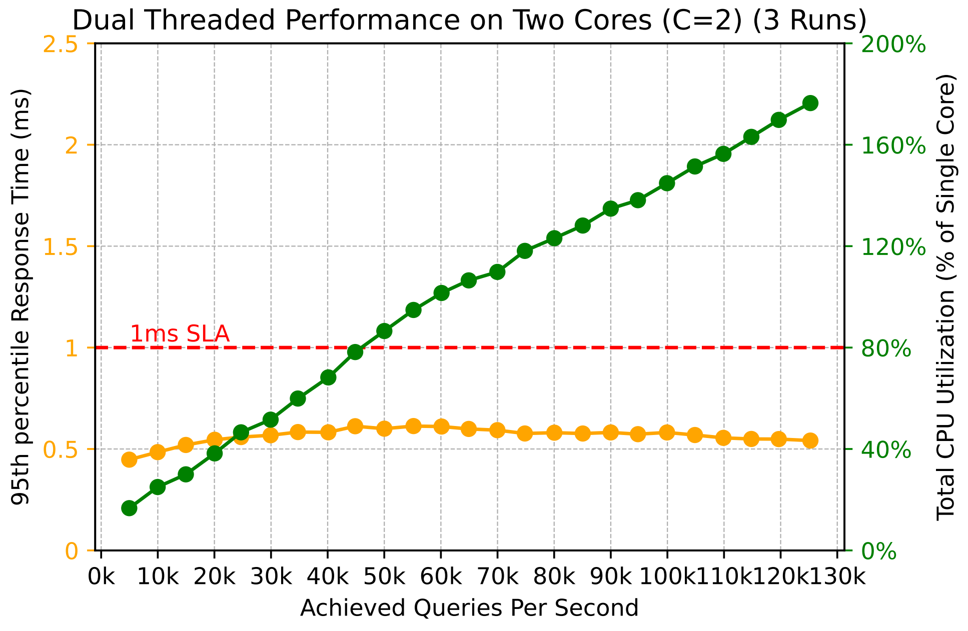
Measure the CPU utilization on the memcached server at each 5-second load time step.

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.

Plot 1 Core:



Plot 2 Cores:



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. **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:
  To guarantee the 1 ms 95th percentile latency SLO, memcached uses two threads and runs on either core 0 or cores 0 and 1. We assign each PARSEC job into one of the following 3 job queues: compute-heavy, memory-heavy and both compute-and-memory-heavy job queue. Core 0 is always executing memcached, core 1 is either running memcached or a PARSEC job and cores 2 and 3 are always running PARSEC jons. Thus, if memcached is not running on core 1, the scheduler picks a job starting from the memory-heavy job queue and if it is empty from the compute-and-memory-heavy job queue and pin it to core 1. For cores 2 and 3, the scheduler picks a job starting from the compute-heavy job queue, then from the compute-and-memory-heavy job queue and finally from the memory-heavy job queue. By pausing and unpausing containers of jobs running on core 1, the scheduler allows memcached job to run on both cores and reverse. Finally, if all the queues are empty but there is a job still running on core 1, the scheduler transfers it to cores 2-3 trying to separate its execution from the memcached job.

- How do you decide how many cores to dynamically assign to memcached? Why?
  Answer:
  Our scheduler uses a simple FSM in order to switch the memcached job between cores. We decided to pin the memcached job either on core 0 or cores 0-1 depending on the CPU utilization of the memcached process. From the graphs in Part 4.1 we can observe that when a single core is used, memcached is violating the SLO when the CPU

utilisation reaches 40%, thus one core is not enough to adhere to the SLO. We took this threshold into account and designed the scheduler so that, when the threshold is reached, `memcached` is allocated a second core, core 1. When the utilisation falls under the threshold, `memcached` is again constrained to using one core, core 0.

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

  Answer: As we mentioned before, PARSEC jobs can be roughly categorised into 3 categories, compute-heavy, memory-heavy, and both compute-and-memory-heavy jobs. The category each job falls into depended on the measurements done in Part 2. Our scheduler's policy is to give higher priority to compute-heavy jobs on cores 2 and 3 as well as to memory-heavy jobs on core 1. A compute-heavy job cannot be executed on core 1, however a memory-heavy job can be executed on cores 2 and 3. More specifically, for cores 2 and 3, the scheduler picks a job starting from the compute-heavy job queue, then from the compute-and-memory-heavy job queue and finally from the memory-heavy job queue. Also, for core 1, the scheduler picks a a job starting from the memory-heavy job queue and if it is empty, it picks a job from the compute-and-memory-heavy job queue. According to each job's category, their CPU needs change as follows.

  – blackscholes: Blackscholes is a memory-heavy job, therefore it doesn't need that much CPU power. Thus, the scheduler wil pin it on core 1, when `memcached` is not using it.

  – canneal: Canneal is a compute-and-memory-heavy job, therefore the scheduler can pin it either on core 1 or cores 2 and 3. It depends on the time all the compute-heavy jobs need to be executed on cores 2 and 3, so that the scheduler will be able to pick a job from the compute-and-memory-heavy job queue. In addition, it depends on the workload since it affects how many times the scheduler will pause a container of a job running on core 0, and as a consequence the time all the memory-heavy jobs need to be executed on core 1, so that the scheduler will be able to pick a job from the compute-and-memory-heavy job queue.

  – dedup: Dedup is a compute-and-memory-heavy job, therefore the scheduler can pin it either on core 1 or cores 2 and 3. The dependencies are the same with canneal job.

  – ferret: Ferret is a compute-heavy job, therefore the scheduler will pin it on cores 2 and 3.

  – freqmine: Freqmine is a compute-heavy job, therefore the scheduler will pin it on cores 2 and 3.

  – radix: Radix is a memory-heavy job, therefore it doesn't need that much CPU power. Thus, the scheduler wil pin it on core 1, when `memcached` is not using it.

  – vips: Vips is a compute-and-memory-heavy job, therefore the scheduler can pin it either on core 1 or cores 2 and 3. The dependencies are the same with canneal or dedup job.

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

  Answer: The number of threads is mapped directly to the number of cores each job utilises. We experimented with the number of threads that the jobs running on two cores use (i.e. instead of 2, we used 4), but we did not notice any speedup, therefore we settled on the direct mapping of threads to the number of cores used. Thus, for a job running on core 1, the scheduler will assign a single thread and for a job running on cores 2-3, the scheduler will assign 2 threads.

  – blackscholes: Blackscholes is a memory-heavy job so, it can be executed either on core 1 or cores 2-3. If blackscholes is executed on core 1, the scheduler will assign it a single thread and if it is executed on cores 2-3, the scheduler will assign it 2 threads. In our experiments, blackscholes is the first job of the memory-heavy queue, thus it runs directly on core 1 and it is assigned 1 thread.

  – canneal: Canneal is a compute-and-memory-heavy job, so it can be executed either on core 1 or cores 2-and-3. The scheduler will assign 1 and 2 threads respectively. As we mentioned before, it depends on the workload and/or the total execution time of all the compute-heavy jobs on cores 2-3.

  – dedup: Dedup is a compute-and-memory-heavy job, so it can be executed either on core 1 or cores 2-and-3. The scheduler will assign 1 and 2 threads respectively. As we mentioned before, it depends on the workload and/or the total execution time of all the compute-heavy jobs on cores 2-3.

  – ferret: Ferret is a compute-heavy job, so it is pinned on cores 2 and 3. Thus, scheduler assigns 2 threads to ferret job.

  – freqmine: Freqmine is a compute-heavy job, so it is pinned on cores 2 and 3. Thus, scheduler assigns 2 threads to freqmine job.

  – radix: Radix is a memory-heavy job so, it can be executed either on core 1 or cores 2-3. If radix is executed on core 1, the scheduler will assign it a single thread and if it is executed on cores 2-3, the scheduler will assign it 2 threads. In our experiments, radix is the second job of the memory-heavy queue, thus it runs directly on core 1 and it is assigned 1 thread.

  – vips: Vips is a compute-and-memory-heavy job, so it can be executed either on core 1 or cores 2-and-3. The scheduler will assign 1 and 2 threads respectively. As we mentioned before, it depends on the workload and/or the total execution time of all the compute-heavy jobs on cores 2-3.

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

  We decided to not collocate more than one job on the same core in order to limit the interference between the jobs, such as cache misses and context switches and offer a better performance overall. Thus, the jobs that run in parallel are those executed on different cores. For example, blackscholes and freqmine run in parallel since blackscholes in pinned on core 1 and freqmine is pinned on cores 2-3. In addition, radix and ferret as well as dedup and canneal run in parallel for the same reason.

- In which order did you run the PARSEC jobs? Why?
  Answer: The scheduler starts executing jobs from the compute-heavy job queue on cores 2-3. Then, when the compute-heavy job queue is empty, the scheduler picks a job from the compute-and-memory-heavy job queue and finally, when both queues are empty, the scheduler picks a job from the memory-heavy job queue and pins it to cores 2-3. On the other hand, when the `memcached` runs only on core 0, there is space for a PARSEC job on core 1. Thus, the scheduler picks a job first from the memory-heavy job queue and if it is empty, it picks a job from the compute-and-memory-heavy job queue. For this reason, the order according to which the PARSEC jobs run is based on the order they are appended into the queues, the execution time of the other jobs as well as the workload of the `memcached` server. Below, we provide an example from our experiments.
  Order (core 1): blackscholes, radix,dedup
  Order (cores 2 and 3): freqmine, ferret, vips, canneal

- How does your policy differ from the policy in Part 3? Why?
  Answer:
  Our policy differs from Part 3 in the sense that it is dynamic. The scheduler uses the data from the previous parts in order to accomodate for the desired SLO. As we can observe from the graphs in Part 4.1, violating the SLO depends on how much percentage of the CPU the `memcached` process uses. Therefore, we have to constantly check if the CPU utilisation of `memcached` is above a threshold (40%) and allocate enough resources. This allows for better utilisation of the cores, since when `memcached` is not stressed, we place a job on core 1, in order to increase the usability of the processor, while not violating the SLO. The scheduler always keeps one core free and exclusive for `memcached` (core 0). Core 1 is alternating between running jobs and `memcached`, according to the CPU utilisation observed during the run. Cores 2-3 are reserved for running PARSEC jobs.

- How did you implement your policy? e.g., docker cpu-set updates, taskset updates for memcached, pausing/unpausing containers, etc.
  Answer:
  In our implementation, we used Python SDK to manage Docker containers as well as the **taskset** command to pin `memcached` job on a specific core or a set of cores. For example, when the scheduler decides to switch `memcached` job from core 0 to cores 0-1 in order to respect the SLO of 1 msec, it uses the following command:

```
sudo taskset -cp 0 | 0-1 <memcached_pid>
```

  We also used **cpuset_cpus** flag in order to pin a container of a job on a specific core or a set of cores. In addition, the scheduler uses Docker **pause()** and **unpause()** functions in order to pause/unpause a container that runs on core 1 because memcached job should be pinned on both cores 0-1 while not losing the job progress of the corresponding jobs. Finally, we used the **update()** function in order to update the cpu_set of a container when the scheduler decides to transfer a container from core 1 to cores 2-3 because all the queues are empty.

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.

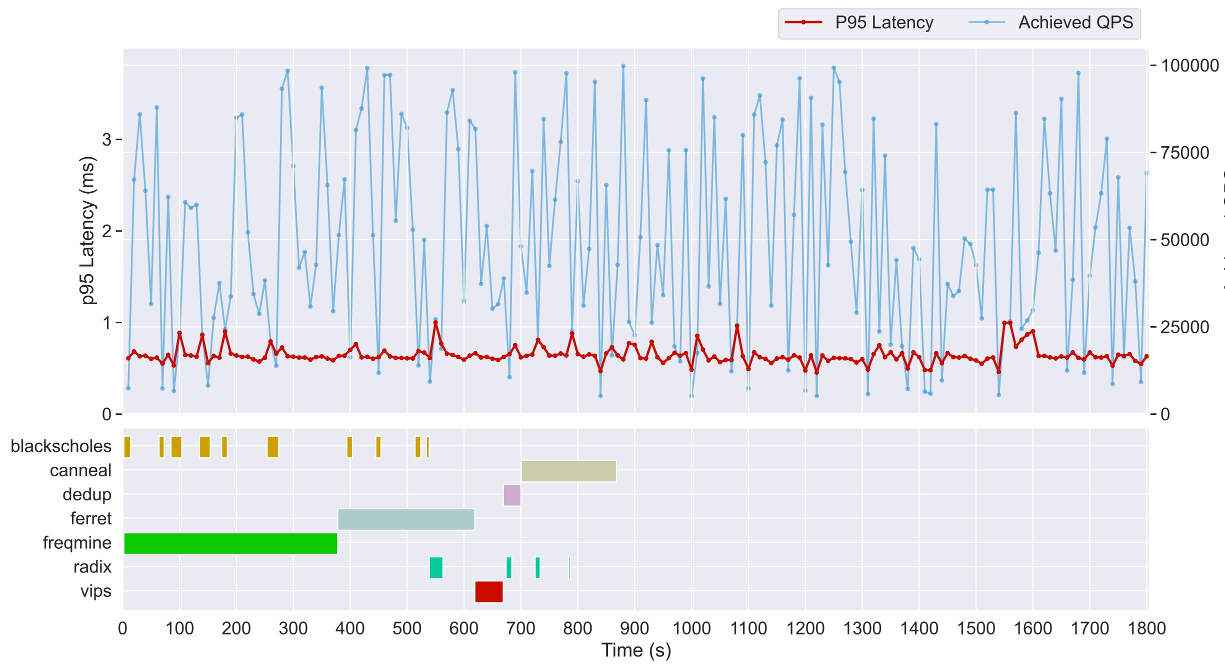
Answer:

job name	mean time [s]	std [s]
blackscholes	121.66	1.15
canneal	174	6.55
dedup	31.66	1.15
ferret	244.33	3.05
freqmine	381	7
radix	46	0
vips	51	1.73
total time	884.66	17

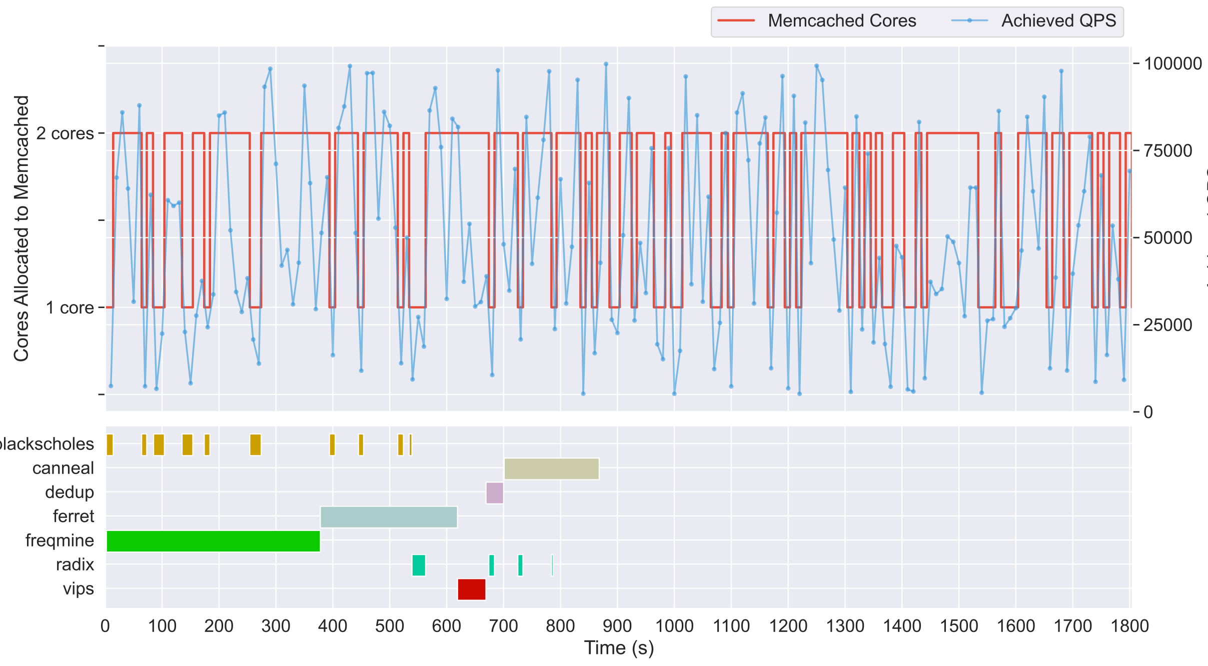
Run	SLO violation ratio (%)
Run 1	0.55 %
Run 2	3.33 %
Run 3	2.77 %

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`).

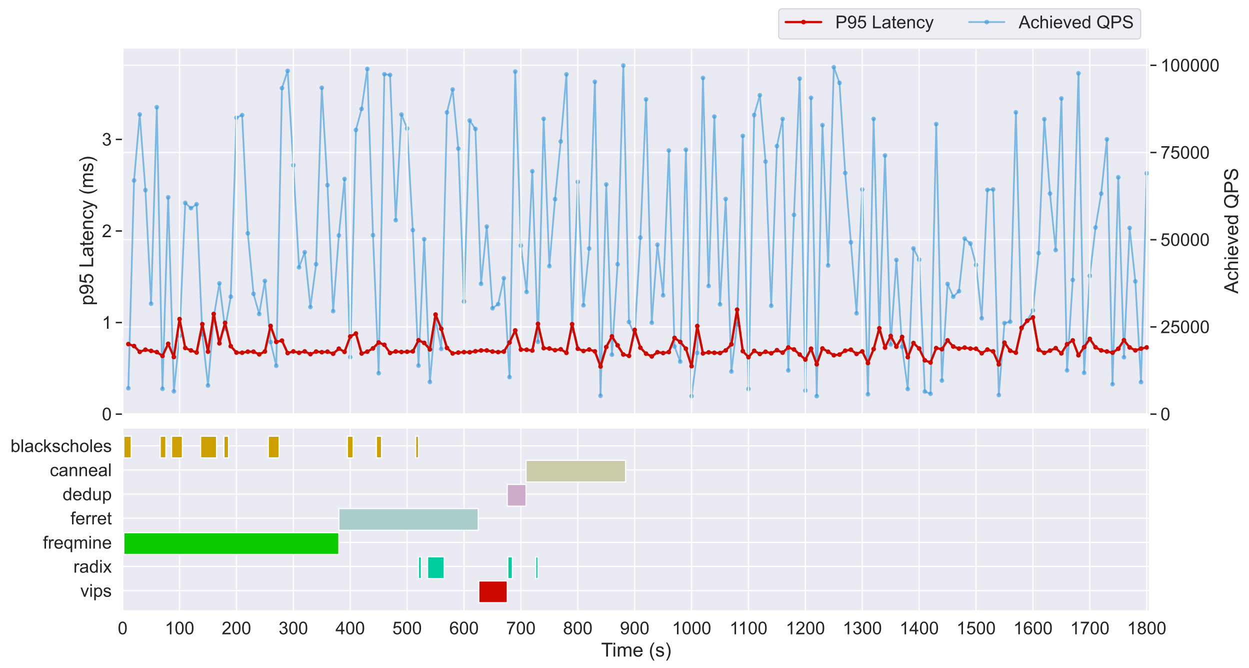
Plot 1a:



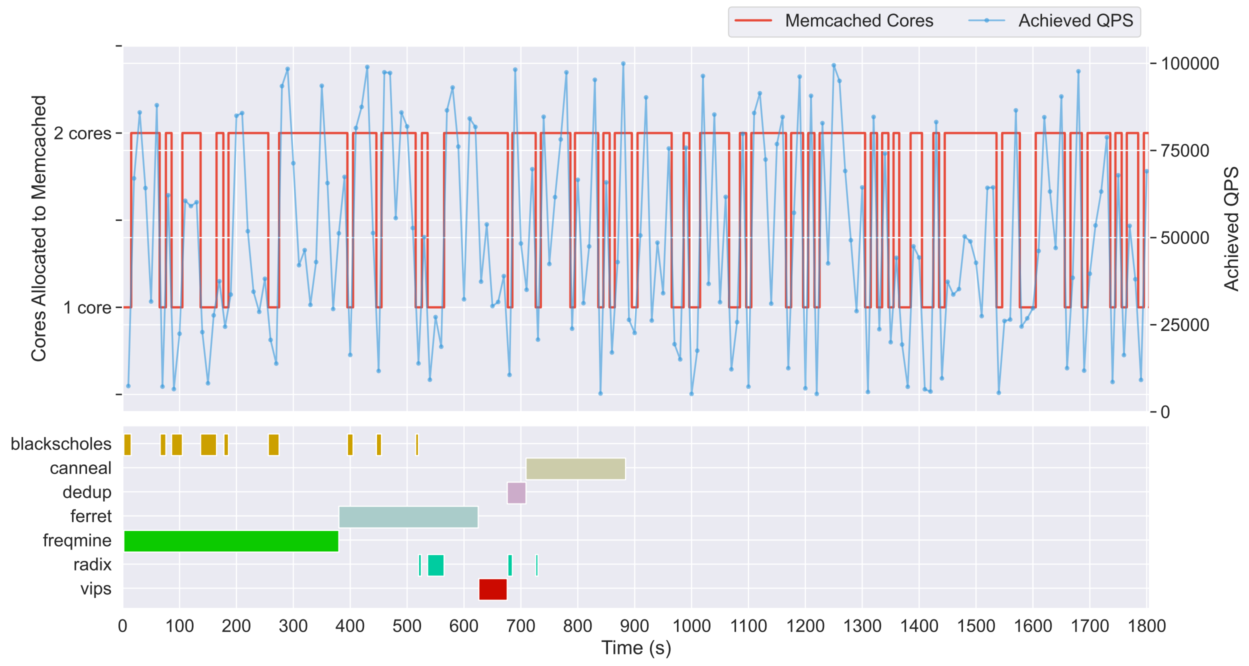
Plot 1b:



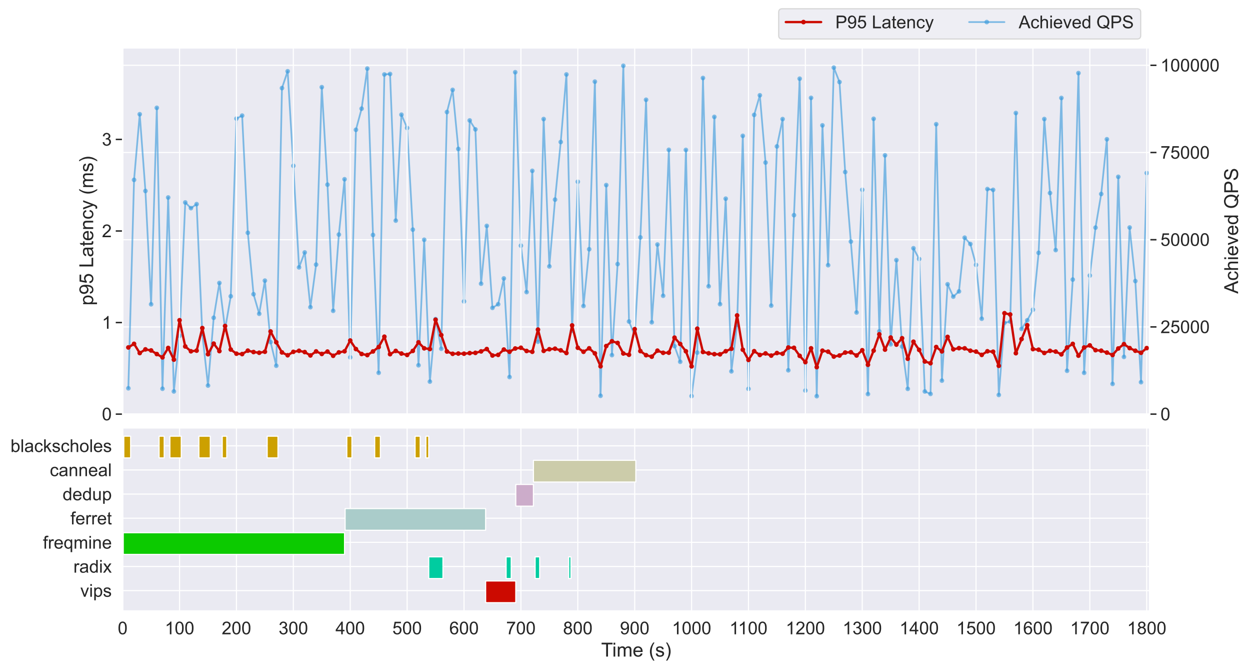
Plot 2a:



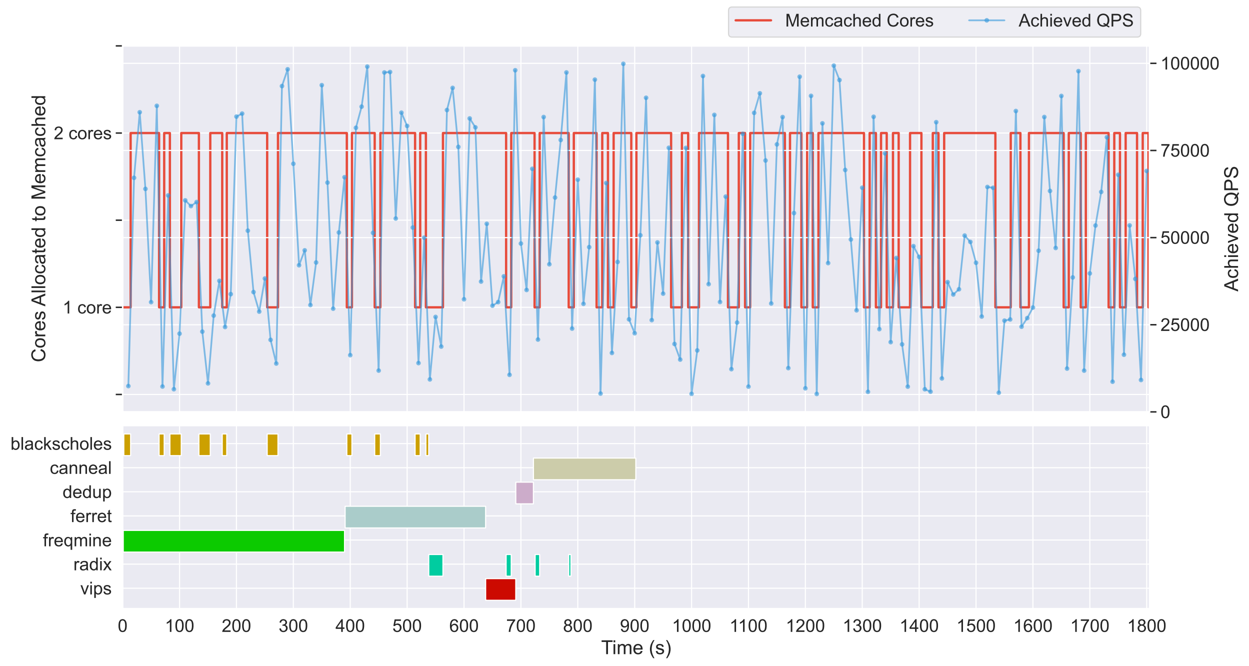
Plot 2b:



Plot 3a:



Plot 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:**
  By repeating the measurements for the new `qps_interval` of 5 seconds, we found out that the SLO violation ratio increases. Upon experimentation, we concluded that if we decrease the utilisation threshold according to which the FSM of the scheduler switches from 1 core to 2 cores and vice versa, the SLO violation ratio will also decrease, but the total execution time will increase. Thus, it should be considered as a tradeoff between execution time requirements and hardware limitations and it depends on the specific user's requirements.

- 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:
  The average SLO violation ratio when `qps_interval` = 5 seconds is 3.42% (average of 3 runs). Since the average SLO violation ratio when `qps_interval` = 10 seconds is 2.21% (average of 3 runs), it seems that our scheduling policy is remains stable when the QPS interval varies. However, we cannot guarantee an SLO violation ratio less than 3% for random `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:
  The smallest `qps_interval` that we can use in order to guarantee an SLO violation ratio under 3% is 7 seconds. More specifically, the violation ratio when `qps_interval` = 7 seconds is 2.71% (average of 3 runs).

- What is the reasoning behind this specific value? Explain which features of your controller affect the smallest `qps_interval` interval you proposed.
  Answer:
  Lowering the `qps_interval` results in more SLO violations. This is caused due to the fact that each time interval will have a different workload. The less the qps_interval, the more duration each workload will have. Thus, a heavy workload with a small qps_interval will have a larger impact on performance compared to a heavy workload with a big qps_interval. The scheduler has to adapt to the new workload by changing

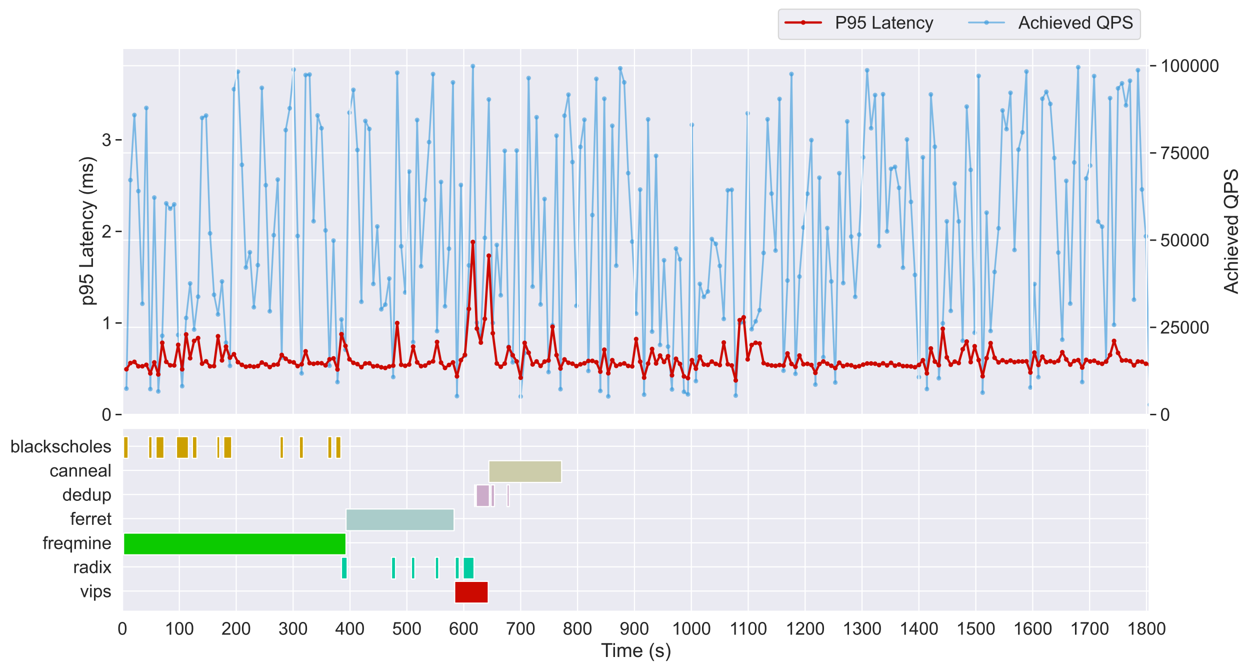
the cores `memcached` runs on, as well as pause or unpause any containers or relocate them entirely on new cores to run. This takes both time, due to context switches and relocations of processes, as well as hardware latencies, such as cached misses. This can have an effect on the performance, since for each interval, the scheduler will take some time to stabilise, while data that may be located on the cache will have to be replaced by new data from `memcached` or a job that has just moved to core 1. These latencies can both be caused by cache misses on working data, as well as, instruction cache misses due to the change of the nature of the process the core is now running.

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:

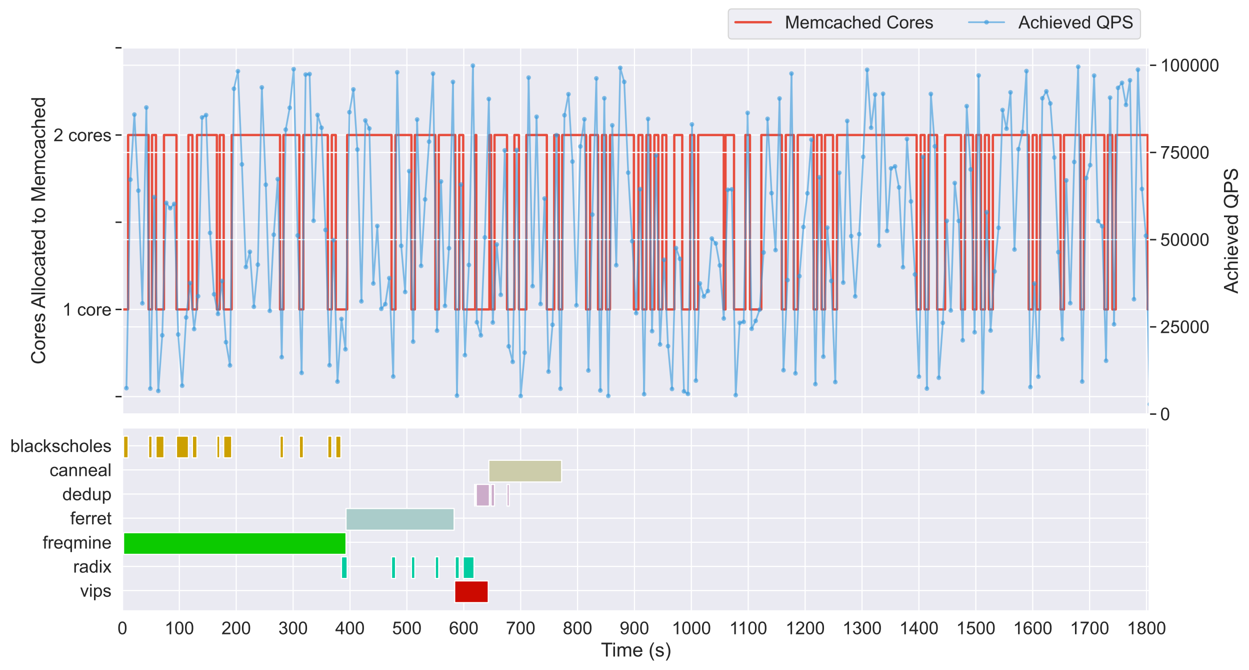
job name	mean time [s]	std [s]
blackscholes	104.66	0.57
canneal	118.33	8.38
dedup	38.33	3.78
ferret	189	1
freqmine	391.33	1.52
radix	55	2
vips	62.33	3.05
total time	764.33	6.8

Run	SLO violation ratio (%)
Run 1	2.32 %
Run 2	3.10 %
Run 3	2.71 %
Average	2.71 %

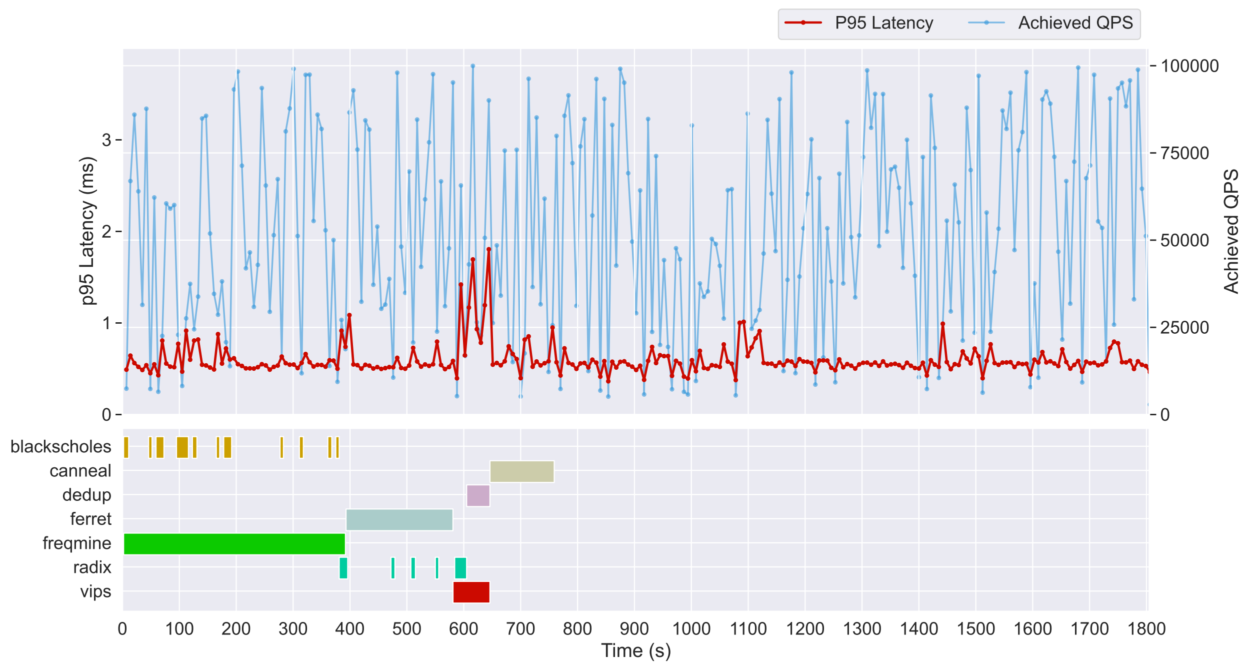
Plot 1a:



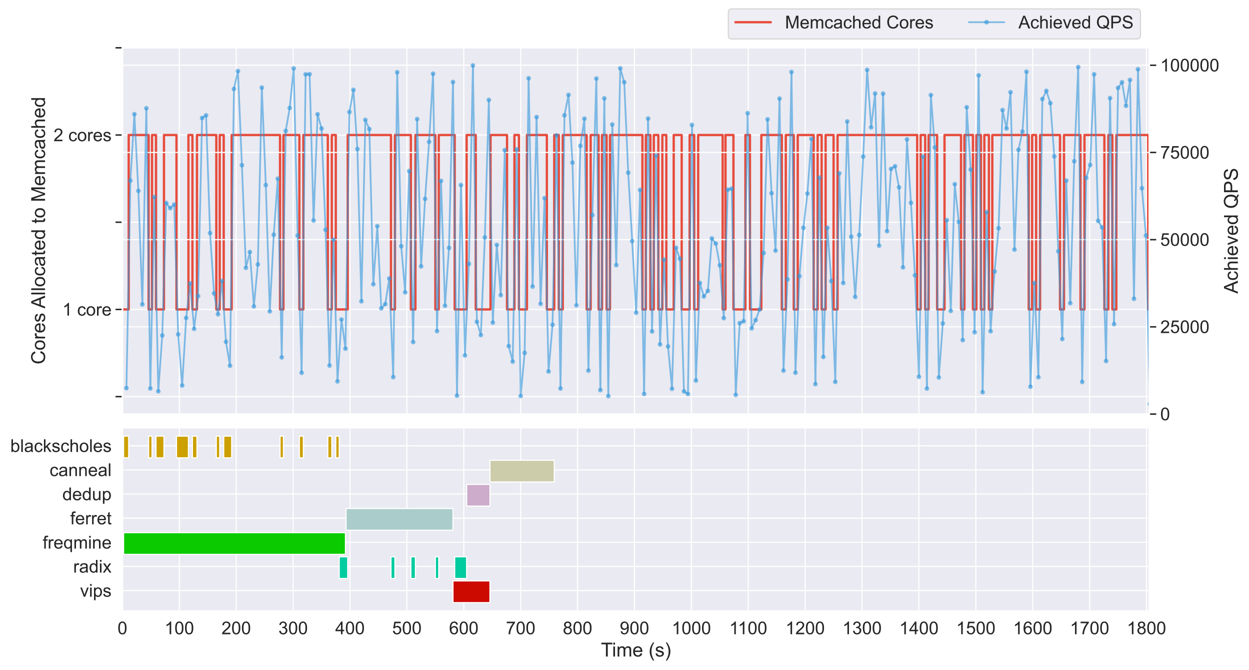
Plot 1b:



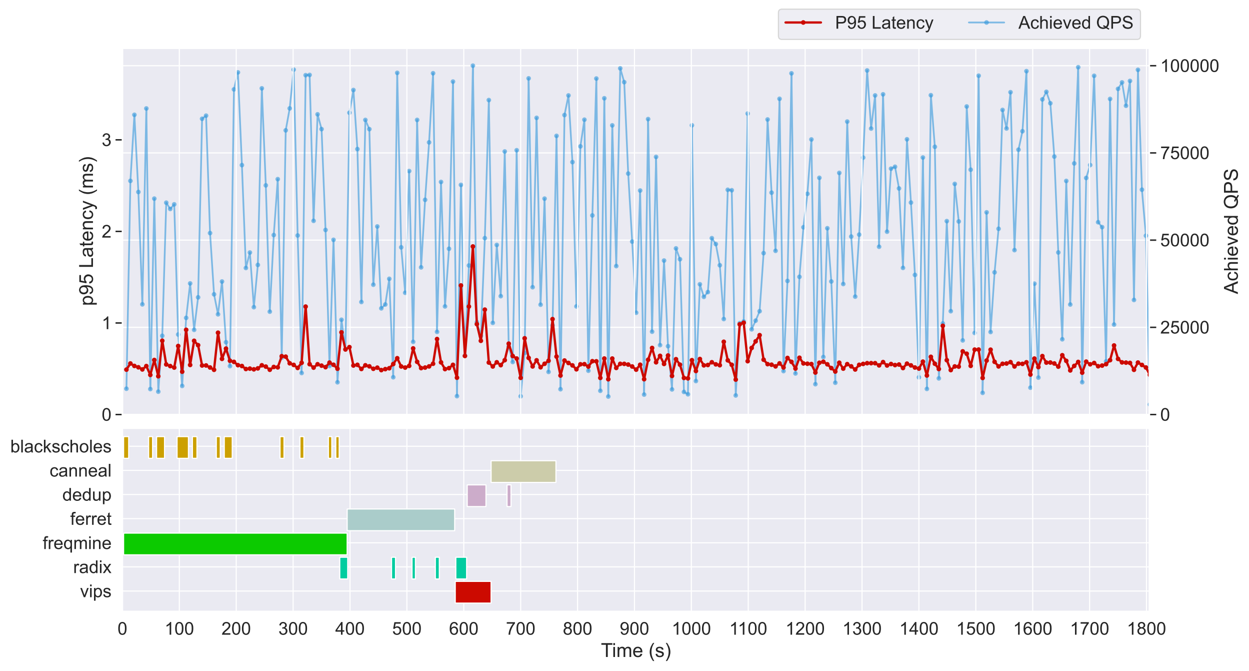
Plot 2a:



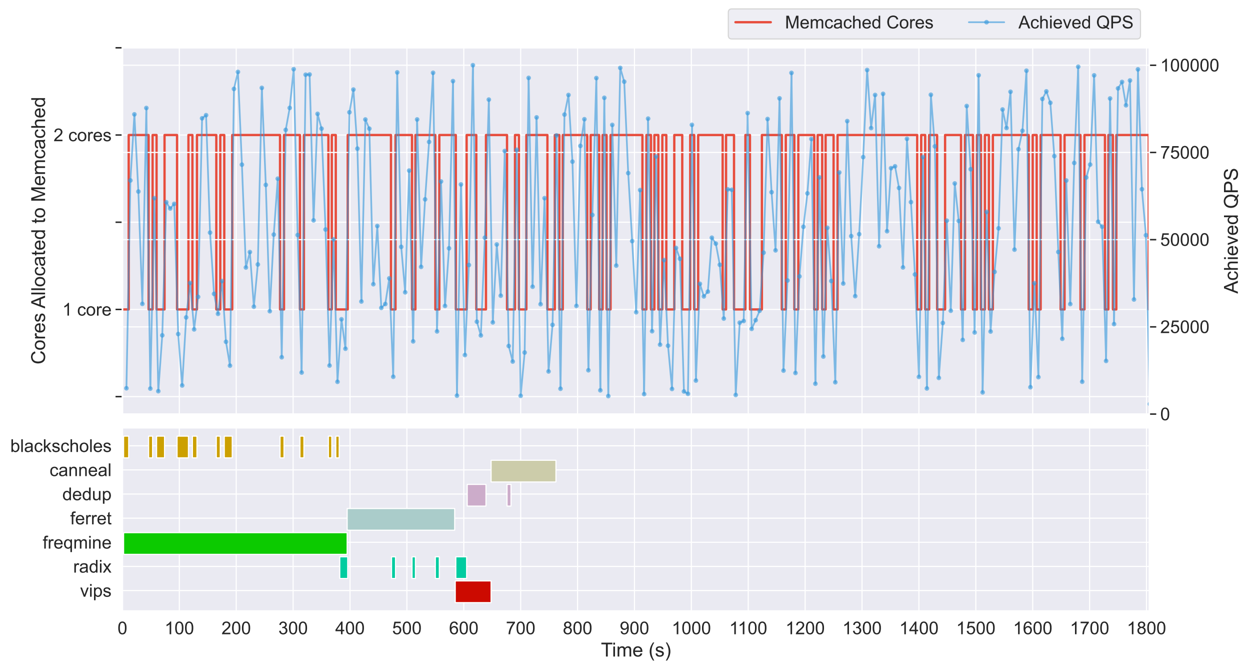
Plot 2b:



Plot 3a:



Plot 3b:


Memcached Cores
Achieved QPS
Cores Allocated to Memcached
2 cores
1 core
Achieved QPS
blackscholes
canneal
dedup
ferret
freqmine
radix
vips
Time (s)