Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Systems@ETH Zürich

Cloud Computing Architecture

Semester project report

Group 39
Peter Abdel Massih - 20-810-214
Leonardo Favento - 22-953-533
Sofija Kotarac - 22-907-331

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: The following table illustrates the SLO violation ratio for memcached across the three different runs:

Number of execution	SLO violation ratio %
Run 1	0.0%
Run 2	0.0%
Run 3	0.0%
Average	0.0%

   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	74.00	0.82
canneal	144.00	4.08
dedup	39.67	0.47
ferret	144.00	0.82
freqmine	136.67	1.70
radix	53.00	0.82
vips	74.33	0.47
total time	149.67	1.25

   Plots: See Figure 1, Figure 2 and Figure 3.

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

   - Which node does memcached run on? Why?

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

Figure 1: The plot shows the p95 latency over time and the duration in which the jobs are executed in the 1st run.

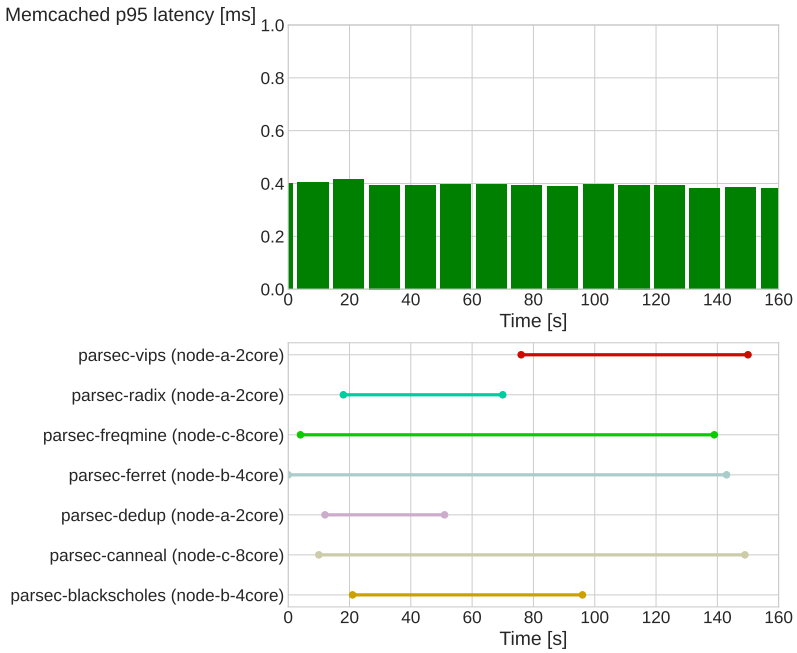

Figure 2: The plot shows the p95 latency over time and the duration in which the jobs are executed in the 2nd run.

Answer: Memcached runs on "node-a-2core" and is the only process that runs on the first core (0). The reason for this is because in part1 we saw that memcached is only affected by interference on CPU and L1i, which are both specific to each core. Thus

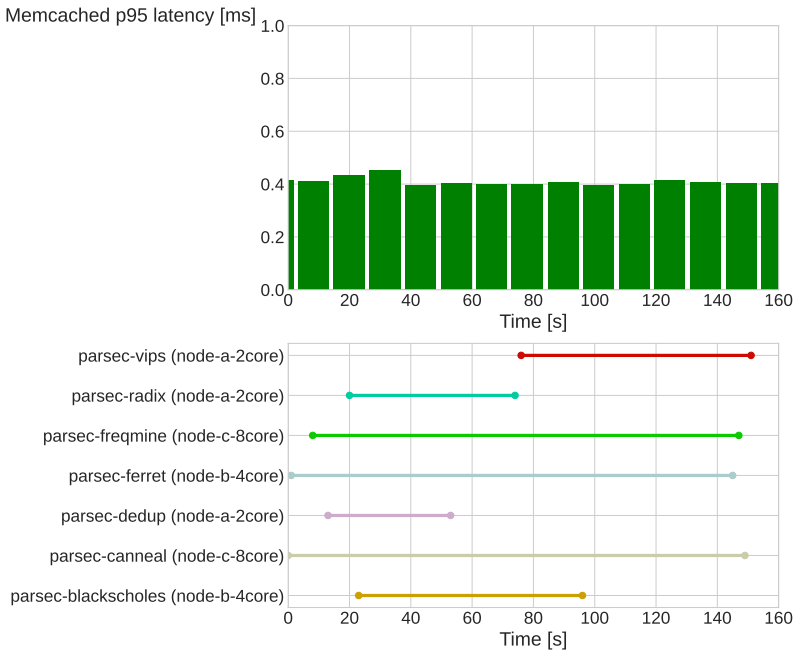

Figure 3: The plot shows the p95 latency over time and the duration in which the jobs are executed in the 3rd run.

in order to avoid interference we run memcached on its own core, which is not shared with any PARSEC job, and minimize interference. Additionally we split the task in 2 threads to parallelize and speedup run-time. Memcached, in our experiments, only needs 1 core to run without violating the SLO, therefore we decided to assign it to the 2-cores node allowing some jobs to run on the same node on a different core, giving maximum potential to the PARSEC jobs to run as fast as possible.

- Which node does each of the 7 PARSEC jobs run on? Why?
  Answer:
  - blackscholes: Blackscholes runs on the node with 4 cores ("node-b-4core"), because based on our analysis in Part 2 of the number of threads one job would benefit the most, blackscholes speedup improves of 60% when running it on 4 threads but less than 25% when upgrading to 8 threads. Assuming that the CPUs our nodes have contain 2 threads per CPU core, the 4 core node is thus enough for running blackscholes. The latter also stresses moderately on L1i, memory, and llc (based on Part 2).
  - canneal: Canneal is running on the node with 8 cores ("node-c-8core"), and based on Part 2 the number of threads that would benefit canneal the most is 4 (as in the blacksholes case) but after running multiple times our scheduler we noticed that the canneal job is the slowest and the bottleneck for our execution. Thus, to get the fastest execution possible we decided to run canneal on 8 threads and although the percentage increase is not significant, the time benefits (due to its slow execution) are serious. 8 threads are needed and canneal is not a resource intensive job stressing mostly on llc (but not very significantly), so we scheduled it on the 8-cores node.
  - dedup: Dedup is running on the node with 2 cores ("node-a-2core"), and according

to Part 2 the number of threads for which this job's speedup would be the most significant is 2 threads. Dedup improves its performance by nearly 70% when ran on 2 threads. The 2 cores node is thus enough for the execution of this job. Dedup stresses mostly on l1i and llc -based on Part 2- and a little on the cpu. This is not a problem in our case where we scheduled memecached on the same node but on a different core (memcached l1i is thus different). Moreover, the memcached application isn't greatly impacted by llc interference. Finally, dedup is also a relatively short job, so it can be scheduled on the 2 threads that are available in the core in which memcached is not executed, while still terminating in a short amount of time.

- ferret: Ferret runs on the node with 4 cores ("node-b-4core") and in resonance with Part 2 the thread number for which this job's speedup would be the most significant is 4 threads (percentage speedup of 75% from 2 to 4 threads). As in the canneal case, this is one of the slowest jobs that could potentially bottleneck, thus we decided to schedule it on 6 threads: in this way there are 2 overlapping threads with blackscholes (as stated before assuming this machine has 8 threads) which don't interfere too much with the execution of the latter. Once blackscholes terminates ferret could benefit fully of the 6 threads. In addition, ferret stresses mostly on l1i and llc and memory, thus could be allocated with blackscholes -where the latter stresses moderately on these components as stated before.
- freqmine: Freqmine is running on the node with 8 cores ("node-c-8core") and based on Part 2 the number of threads that would benefit freqmine the most is 4 (linear increase from 2 to 4 threads) but after running multiple times our scheduler we noticed that the freqmine job is one of the slowest in its execution. Thus, to get the fastest execution possible we decided to run freqmine on 8 threads (the 8 core node has 16 threads based on our assumption), and although the percentage increase is not significant, the time benefits (due to its slow execution) are serious. This job stresses mostly on l1i, cpu, llc and memory, thus can be collocated with canneal which is not a very stressful job on these components.
- radix: Radix runs on the node with 2 cores ("node-a-2core") and conforming to Part 2 the number of threads that would benefit radix the most is 8 threads (nearly 50% speedup from 4 to 8 threads). Since radix is a very short job, and benefit from a linear increase (100% speedup) when going from 1 to 2 threads, we decided to allocate it on the 2 cores node. Radix (like canneal) is not a resource hungry job stressing only slightly on llc. Another reason to co-locate it with dedup and memcached, is its short execution time. As stated before for the dedup job, radix doesn't interfere with the main components that memcached rely on making it a good match on this node.
- vips: Finally, vips is running on "node-a-2core" - the 2 cores node. According to our analysis in Part 2, vips would best benefit from running on 4 threads (nearly a linear speedup from 2 to 4 threads). Since vips is relatively short and also has a linear increase when upgrading it from 1 to 2 threads. We decided to schedule it on the 2 cores node alongside dedup, radix and memcached (on a different core). It is a quite resource hungry job (mainly l2, l1i and llc), but its short execution time outweighs its heavy resource usage making it a good match on this node.

- Which jobs run concurrently / are colocated? Why?
  Answer: PARSEC jobs **dedup**, **radix** and **vips** all run on the second core of "node-a-

2core" in 2 threads. Since they all have short running time, they can be co-located safely while still finishing all of their executions in a total time comparable to the running time of the jobs that require most time to complete (freqmine, ferret and canneal). Then PARSEC jobs **ferret** and **blackscholes** both run on "node-b-4core" split up across all cores in 6 and 4 threads, respectively. The reason they are colocated on node-b is because in Part 2 we found that **ferret** suffers from l1i, llc and memBW interference, whereas **blackscholes** only interferes slightly with these components, so it would not add much additional stress and the two jobs won't affect negatively the performance of the other. Finally, PARSEC jobs **canneal** and **freqmine** are colocated on "node-c-8core" each one split into 8 threads across the 8 cores. The reason we do so is because in Part 2 we found that **canneal** adds little to no stress on the cpu, l1d, l2 and memBW, so **freqmine** can share these same cores since it suffers major interference on cpu and l1i and slightly memBW. The minimal impact on the resources of **canneal** guarantees that the performance of **freqmine** will not suffer additional stress and vice versa, which is optimal considering that both jobs are two of the most time expensive jobs.

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

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

  Why: After assigning each job to specific nodes and threads, we started by testing the execution of the jobs in their default order (blackscholes, canneal, dedup, ferret, freqmine, radix, vips) and we noticed that in most executions the bottleneck would be ferret, being the last job to finish, while starting only about 20 seconds after the start of the first job being executed. This happened because since ferret was scheduled in the same node as blackscholes (whose container creation requires a significant amount of time) it was put on hold until the latter started running. Therefore, we decided to switch the order in which ferret and blackscholes PARSEC jobs are scheduled, so that ferret could start its execution earlier (which means that it can finish earlier) and since blackscholes' running time is relatively short, neither of these two jobs would be bottleneck of the running time as a whole. After this change the bottleneck of the total execution becomes vips, which in our configuration gives the best performance when it's scheduled as the last job (as in the default order), since its container creation requires a noticeable amount of time and therefore would affect the performance of the other jobs running on the same core (dedup and radix), so no other changes would improve the total runtime.

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

  Answer:

  - blackscholes: Blackscholes uses 4 threads since as we saw in Part 2, its performance increases significantly only when increasing the number of threads in which it is executed up to 4. Furthermore, it is a job that runs comparatively faster than ferret (which is the other job executed in the 4-cores node), so adding additional threads wouldn't produce any benefit, since it could affect negatively the performance of ferret while not obtaining a worthy improvement in its own performance.
  - canneal: Canneal runs on 8 threads since its running time is generally the largest across different tests executed. Therefore, even if the increase in performance obtained by upgrading the number of threads of its execution is less noticeable starting

from 4 threads on, it is necessary in order to decrease the total running time, since otherwise it would be a bottleneck.

- dedup: Dedup runs on 2 threads since as we saw in Part 2, its performance doesn't increase significantly when upgrading to more than that amount of threads (obtaining even a decrease in performance when running on 8 threads). Moreover, its execution time is short so executing it on 4 threads would provoke more harm (by congesting the available threads) than good.
- ferret: Ferret runs on 6 threads since, even if in Part 2 we observed that its execution time doesn't improve noticeably when running on more than 4 threads, it is one of the jobs taking the biggest amount of time. We decided to use 6 and not 8 threads since the node (node-b-4core) contains 4 cores (thus it can execute 8 threads at the same time), and blackscholes (which is a job that runs quite fast, but benefits noticeably when running on 4 threads) also needs to be executed in the same node, therefore running ferret on 8 threads could potentially interfere significantly on the performance of blackscholes.
- freqmine: Freqmine is one of the jobs that takes the most amount of time to execute. Its performance improvements are significant when the number of threads it runs on goes up to 4 but in order to avoid it being a bottleneck we decided to run it on 8 threads in the same node - with 8 cores - as canneal, which is also running on 8 threads. Thus, both of them have always 8 threads available at each moment of their execution.
- radix: Radix performance increases significantly when running on 8 threads, but since it is a job that runs in a short amount of time even when it's executed on a lower amount of threads, we decided to run it on 2 threads (on the core next to the one on which memcached is executed). Like in the case of dedup, running it on more than 2 threads and on more than 1 core, would have caused less threads available to other processes (in this case memcached, which would have violated the SLO).
- vips: Vips runs on 2 threads since, even if it would best benefit from running on 4 threads, it is has a relatively short runtime and also has a linear increase when running it from 1 to 2 threads. Moreover, since running it on the 4 cores node or on the 8 cores node would negatively affect the performance of the most intensive jobs, we decided to schedule it on the 2 cores node, where running it on more than 2 threads wouldn't improve its performance due to it being executed only on 1 core.

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

  Answer: We modified the .yaml configuration files of memcache and of the 7 PARSEC jobs. For memcache, we specified the node in which it should run with the `nodeSelector` Kubernetes feature, and the core of the node in which it should run with the taskset command, as well as the number of threads of the execution. For all the jobs, we specified the node of the execution (again with Kubernetes' `nodeSelector`) and number of threads on which the jobs would run. For the jobs running in the same node as memcache, we additionally specified the core on which they would run through taskset, while for all the others we relied on the operating system scheduling policy.

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

Answer: We found the taskset command useful when choosing cores in the node-a-2core node that is running memcached to reduce interference on the l1i and CPU - components where memcached execution rely on the most. For the other jobs, specifying the cores of their execution led to worse performance compared to leaving the scheduling to the operating system at run-time, since allocation in the latter case will be based on the lowest numbered RTSS (monitoring application providing statistics on cpu usage) processor available in the system. Manually selecting the cores, instead, can lead to not full usage of the CPU cores at all times.

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:

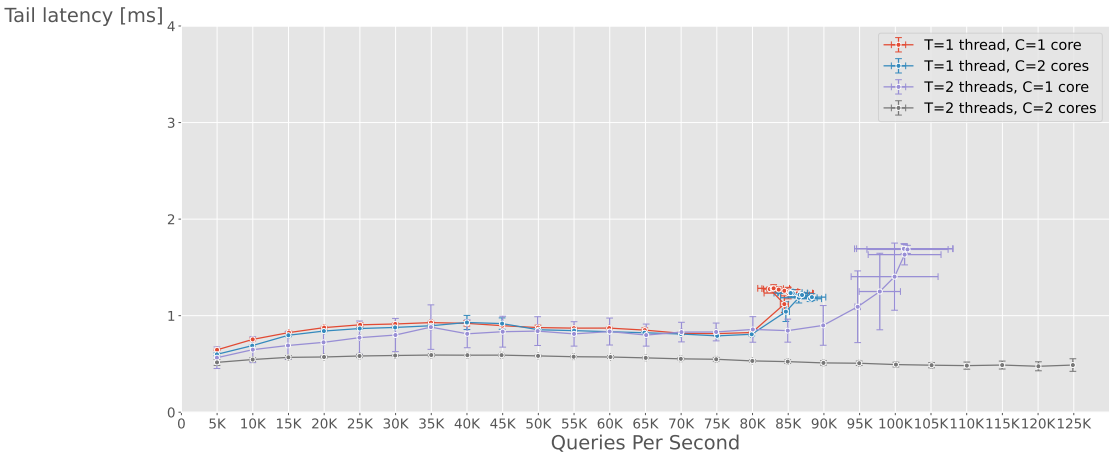

Figure 4: The plot shows how memcached preforms in various combinations of thread distributions and core allocations across 3 different runs, where error bars represent the standard deviation from the mean. The performance is characterized by tail latency according to varied loads on memcached service.

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: According to Figure 4, memcached performance over 1 thread spread across one or two cores achieves similar tail latency: it remains under $1ms$ for QPS under 80K, and increases to reach an apogee at $1.2ms$ over that threshold. On the other hand, when running

memcached with 2 threads on 1 core the tail latency stays below $1ms$ up until 35k QPS; after that point the 95th percentile averages below $1ms$ for higher QPS but for some runs the values could go beyond $1ms$, reaching a peak of around $1.8ms$ at 100K QPS. Finally, when the 2 threads are spread across 2 cores, the latency is nearly constant at a value around $0.5ms$ for all the achieved QPS.

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: As we can observe in the graph obtained by running memcached across different configurations of threads and CPU cores, the only way to support the SLO of 1 ms of latency at 125K QPS is to run memcached on 2 threads and on 2 CPU cores. For all the other tested configurations in fact, it wasn't possible to reach 125K QPS.

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: As said in the previous answer, the only configuration that supports the 1 ms 95th percentile latency SLO is the one that executes memcached on 2 threads on 2 CPU cores. Therefore, if the number of threads is fixed when launching memcached, we need to run it on 2 threads: until around 80K QPS 1 core will be sufficient, after that point 2 cores will be necessary not to violate the 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: See Figure 5 and Figure 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
```

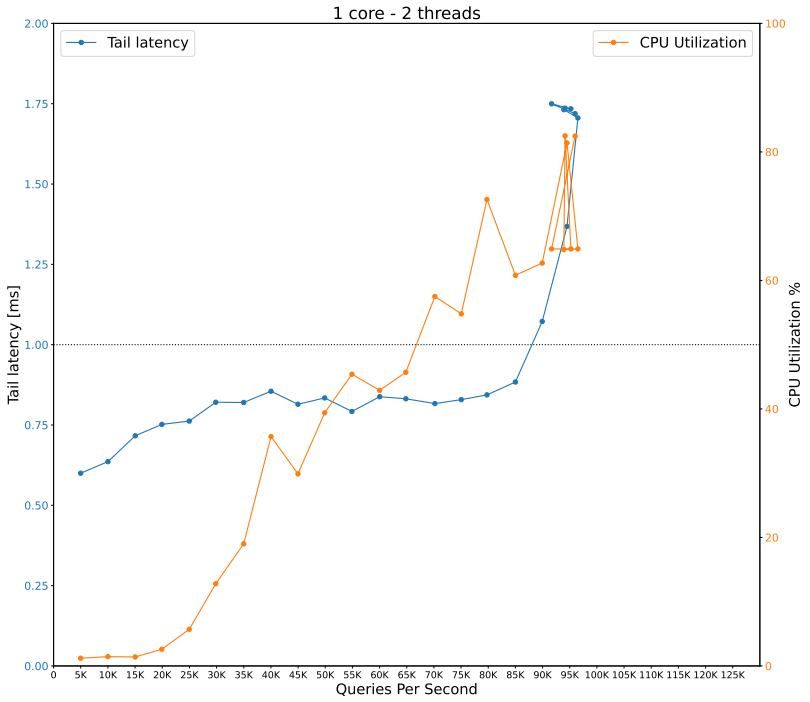

Figure 5: the plot shows both CPU utilization and tail latency based on the load placed on memcached service when it is running on 1 core across 2 threads.

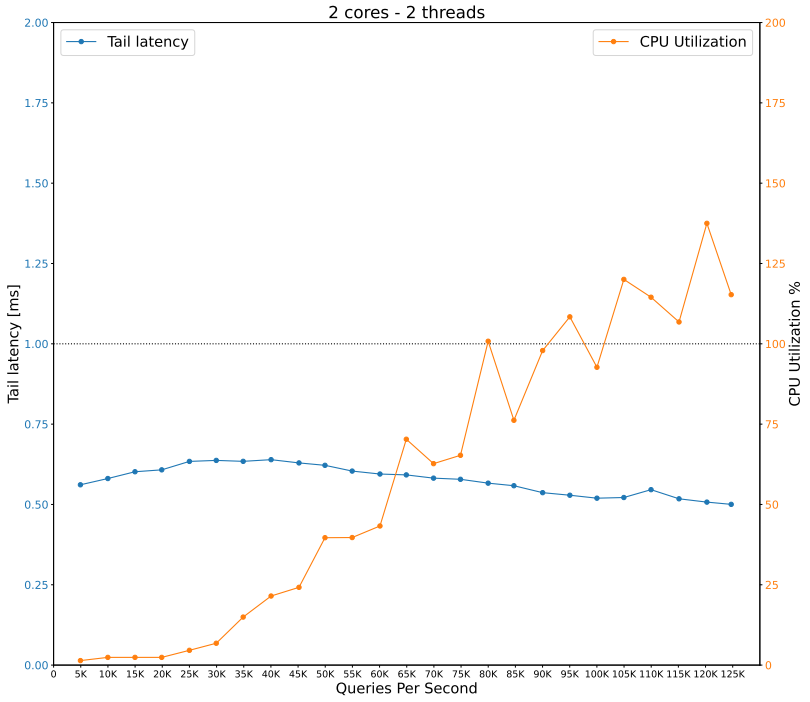

Figure 6: the plot shows both CPU utilization and tail latency based on the load placed on memcached service when it is running on 2 core across 2 threads.

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: Our scheduling policy has two main parts: first, the Memcached service process can switch from running on 1 to 2 cores or vice-versa depending on the CPU utilization in the assigned cores where Memcached is running. This transitioning occurs when the CPU utilization reaches a set threshold of 60%. To address the unexpected increase in CPU utilization when transitioning from 2 to 1 core, an anti-burst policy was implemented to maintain stable and low CPU utilization before the switch. Second, our job scheduler instantiates two queues, one for long jobs and one for short jobs ensuring that two jobs are always running simultaneously. Associating one short job with one long job minimizes interference and increases parallelization. If all short jobs have completed, our policy certifies that 2 long jobs or one job is always running.

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

  Answer: We run Memcached on two threads while assigning dynamically one or two cores to its execution. The shift from 1 core to 2 cores occurs, based on answer 1.d, as soon as the CPU utilization on core number "0" is greater or equal than 60%. On the other hand, we observed an unexpected rise in CPU utilization when migrating from 2 cores to 1 core. This caused a back and forth transition between the two assignments, potentially leading to a violation of the SLO due to a peak in CPU utilization. Therefore, to handle this unsolicited phenomena, we have implemented a precautionary measure. Before we downgrade from 2 cores to 1 core, we ensure that the CPU utilization has been continuously under the threshold of 60% on cores numbered "0,1" for at least 2 seconds (8 loops iterations of 0.25 seconds each). This way we make sure that the utilization is stable at low values, preventing any sudden spikes after the alteration.

  All in all, we run memcached on 1 core as much as we can, so that the jobs have more resources available and thus can run faster while not violating the SLO, and switch to 2 cores to prevent the latency to go over 1 ms whenever the QPS load gets higher.

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

Answer:

- blackscholes: For both instances when memcached is running on one core or running across two cores (numbered 0,1) we assign only one core (numbered 2) for blackscholes. We do this because blackscholes is a shorter job and thus can finish faster. Since we want to optimize the runtime of all our PARSEC jobs, we do not need to allocate many resources for shorter jobs such as blackscholes, instead we give more cores to longer jobs. Moreover, blackscholes is less resource-intensive and has lower interference, which can be seen in part2 (Interference Behaviour Table), it can be collocated with other jobs running on the same core.
- canneal: Canneal is a long job and thus needs more time and resources during execution. Since our goal is to minimize runtime for PARSEC jobs, we prioritize assigning more resources to long jobs with slower execution. As such, when memcached is running only on one core (numbered 0) we assigned 3 cores (numbered 1-3) for canneal in order to provide it with more resources and computing power for a faster runtime. When memcached runs across two cores (numbered 0 and 1), we assign 2 cores for canneal (numbered 3 and 4) in this way we maximize resource allotment and computing power for a faster execution.
- dedup: Similar to blackscholes, dedup is also a shorter job and thus we assign it to one core (numbered 2). Since dedup is short and already finishes fast, we do not need to assign it further resources (which are needed for longer jobs to improve speed).
- ferret: Similar to our reasoning for above-mentioned canneal, ferret is a long job which requires more resources during execution. When memcached is running on one core (numbered 0), we assign three cores (numbered 1-3) for ferret; when memcached is running across two cores (numbered 0 and 1) we assign two cores for ferret execution (numbered 2-3).
- freqmine: Similar to above mentioned canneal and ferret, freqmine is a long job which requires more resources during execution. In order to lower its runtime, we maximize resource allotment and computing power by assigning 3 cores (numbered 1-3) for freqmine execution, when memcached is running on one core (numbered 0). When memcached is running across two cores (numbered 0 and 1) we assign two cores for freqmine execution (numbered 2-3).
- radix: Since radix is a shorter job, it runs faster than longer jobs would. Additionally, radix is less resource-intensive and causes less interference than longer jobs, as seen in part2 (Interference Behaviour Table). Due to this, it would be a waste of computing power and resources to allocate many resources to radix, especially when compute-heavy and resource-heavy jobs need them more. Thus, we do not need to allocate many resources for radix. In both instances when memcached runs on one core or two cores, we assign radix to execute on one core (numbered 2).
- vips: Vips is a shorter job (like radix, blackscholes and dedup) and thus for the same reasoning mentioned above, in both cases of memcached cpu core allocation, we only allocate one core (numbered 2) for vips' execution.

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

  Answer:

  - blackscholes: Based on our results from part3 (where the available nodes could allow up to 8 threads), now we scheduled based on different assumptions: we assume

that when memcached runs on one core (numbered 0) we have 3 available cores (numbered 1-3). The reason is that we want to maximize the jobs' performance when the memcached process utilizes the least amount of resources (1 core). We also assume that each of these cores has a capacity for 2 threads. Thus in the 3 cores availability case, if we exhaust the computing capacity we can run a total of 6 threads under these assumptions. In part3 blackscholes utilizes 4 threads, we now adjust to smaller thread capacity and reallocated blackscholes to 2 threads. This is because blackscholes is a short job, and we have seen in part3 that by adding parallelization across more cores would not improve it's runtime (which is already fast). By doing this, we save 4 additional threads that might be useful for longer jobs that could cause bottleneck without sufficient parallelization. This fits perfectly with our scheduling policy, since we schedule blackscholes to run concurrently with ferret which executes on 4 threads across cores 2,3 or cores 1,2 and 3.

- canneal: *As stated above we assume a best-case scenario where memcached runs on one core numbered 0, and each core has 2 thread capacity.* We saw in part3 that canneal's runtime is the largest so it should be allocated more threads to allow for optimized runtime with parallelisation. Previously we split canneal across 8 threads to achieve this, however now we execute canneal as 4 threads. Canneal can also run in parallel with a short job running on 2 threads, and can thus exhaust all thread capabilities while favouring the long job. In this particular case we wish to run canneal simultaneously with 2 jobs sequentially: first vips, and then after vips has executed we launch freqmine. Vips runs across 2 threads and freqmine runs across 4 threads thus we have a minimum of 6 threads and a maximum 8 threads running simultaneously.
- dedup: *As stated above we assume a best-case scenario where memcached runs on one core numbered 0, and each CPU core has 2 thread capacity.* Referring back to part2 and part3, we know that dedup is a short job and previously ran on 2 threads. Since as we saw in Part 2, its performance doesn't increase significantly when upgrading to more than that amount of threads (obtaining even a decrease in performance when running on 8 threads) and we have 6 thread capacity, we can afford to reduce dedup execution on a single thread without significantly affecting its runtime. With this, we provide 5 threads for other (longer) processes. This fits with our scheduling policy which runs dedup concurrently with ferret (which runs on 3 threads).
- ferret: *As stated above we assume a best-case scenario where memcached runs on one core numbered 0, and each cpu core has 2 thread capacity.* Referring back to part2 and part3, we know that ferret takes longer to execute. Since we want ferret to be able to run parallel with dedup, blackscholes, radix and vips, which all run on core 2 across at least 1 thread and at most 2 threads, we allocate 3 threads for ferret. That way if all aforementioned processes run simultaneous (5 threads) we have not over-exhausted thread capacity.
- freqmine: *As stated above we assume a best-case scenario where memcached runs on one core numbered 0, and each core has 2 thread capacity.* We know from part3 that freqmine is one of the longest executing jobs, as such we want to split it across many threads for optimal parallelisation and allow for increased runtime. Thus we parallelize it on 4 threads, since splitting it into less threads would significantly decrease runtime performance.

- radix: *As stated above we assume a best-case scenario where memcached runs on one core numbered 0, and each core has 2 thread capacity.* Since radix is a short job with fast execution time, we decide to run it as a singular thread. Since dedup (running as a singular thread) finishes executing before we launch radix, core 2 has freed up 1 thread from dedup and can now support running radix. Additionally, since we scheduled radix and ferret to run simultaneously, and ferret executes on 3 threads, this combination allows for both processes to run concurrently whilst maximizing runtime.
- vips: *As stated above we assume a best-case scenario where memcached runs on one core numbered 0, and each core has 2 thread capacity.* Vips is a shorter job and thus needs less time and resources to execute. From part3 we found that vips has a linear increase when running it from 1 to 2 threads. Additionally, in part2 we can see that it is a resource intensive job (mainly l2, l1i and llc). For these reasons, we decided to maximize its parallelization on core 2, by splitting it across 2 threads. This leaves us with 4 threads across cores 2-3. For the first half of vips' execution it runs concurrently with ferret which requires 3 threads. Once ferret completes, we launch canneal which requires 4 threads across cores 2-3. By allocating 2 threads for vips on core 2, we guarantee ferret and canneal can run concurrently with sufficient parallelization.

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

  Answer: At each moment, there will be 2 jobs running concurrently (until there is only one job left). According to our scheduling policy, those 2 jobs will be extracted from different queues: one from the queue containing long jobs (canneal, ferret, freqmine) and one from the queue containing short jobs (blackscholes, dedup, radix, vips). Short jobs will always be collocated on core number 2, while long jobs will shift from cores 1,2,3 to 2,3 and vice-versa depending on the cores in which memcached is running on a given moment (so that no job will never be collocated with memcached). Since the short jobs queue will be emptied first, after all the short jobs will have terminated their execution, there will be 2 long jobs running concurrently, both on cores 1,2,3 or 2,3 (again depending on where memcached is running). The reason for which we execute only 2 jobs at most at a time is that in this way we avoid high interference (which would be caused by more jobs running concurrently), while still benefiting from parallelization. According to part2, ferret - a long job - is strongly affected by interference, thus scheduling a short job with low interference alongside it, such as radix and blackscholes, doesn't reduce its performance. The other jobs that run concurrently with ferret are dedup and vips, which both highly interfere with ferret's resources. Dedup being very short doesn't affect ferret for a long period, and by the same argument vips being scheduled alongside ferret only when the latter execution is nearly complete, their execution will be concurrent only for a short time. As a result, their concurrent execution is brief, minimizing any significant impact on performance and the overall completion time. Moreover, after ferret is completed, canneal will be the next long job scheduled: since it is not resource intensive, the execution of the jobs running concurrently to it won't be affected. Finally, freqmine will run simultaneously to canneal, where the latter as stated before isn't a job that strongly interfere, consequently freqmine will not be significantly delayed.

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

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

Why: As described in the answers above, the order of the execution of the jobs is defined by two queues, with 2 jobs running concurrently at each time. The order therefore can potentially change across different executions, although in our tests it has been consistent. Ferret will be the first job running, being extracted from the long jobs queue. In parallel to ferret, the 4 short jobs will run one after each other, since their execution will complete before the one of ferret. In case ferret finished before the start of one of those short jobs (which never happened in our tests), canneal would have preceded one or more of them. After ferret terminates its execution, the only jobs left to run will be in order canneal and freqmine, extracted from the long jobs queue.

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

  Answer: In part4, we do not have the extensive choices of nodes with different number of cores as in part3. Our current policy, therefore, can not benefit from running all the jobs on different nodes with different resources as we had the choice in part3, due to the limitation to one 4 cores node. In part3 we used a static scheduling task without getting information at run-time and the QPS load from the clients was nearly constant around 30k QPS, whereas now we assign the cores dynamically at running time based on the current parameters such as CPU utilization and the QPS load is now random in each interval. Since we have limited resources in this part, we can not just statically allocate enough resources for memcached while respecting the SLO for any QPS load (and thus CPU utilization) in the given QPS bounds, and at the same time optimizing the total running time of the parsec jobs. All in all, the jobs and the Memcached server are now scheduled based on resource readiness instead of a predetermined collocation strategy.

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

  Answer: To implement our policy, we used the `taskset` command to select the cores in which memcached should run, by executing it at runtime from our scheduler. To manage the dockers, we used the Docker SDK for Python, which calls the docker commands: the `create` command to create the containers, the `start` command to start the scheduled containers, the `update` with `cpuset_cpus` parameter to change the cores on which the jobs are running and the `remove` command to remove exited containers. In our implementations, we didn't make use of the pausing/unpausing feature of docker, since we run at most only 2 jobs at a time and we wait for them to finish before scheduling new jobs. Moreover, pausing and unpausing jobs leads to unwanted timing overheads which would only increase the total running time.

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

  Answer: While designing our scheduler, we were always conservative when it came to allocating resources for the memcached process. For instance, in the special case where we transition from 2 cores to 1 core for the running memcached server, we always make sure we achieved a somewhat stable low incoming load before switching to a lower number of cores. This check is done based on a formula obtained through empirical results over which we applied an exponential regression to get hyper-parameters granting us a reliable threshold even for low QPS intervals and good performance in the case of higher QPS intervals (and thus more stable loads). The aforementioned makes sure the

transitioning can occur safely with no sudden spikes (if it did spike on the next iteration we would instantly switch back to 2 running cores) or to be protective and avoid violations of the SLO. Finally, as stated before to be on the safe side when running jobs alongside memcached (on the same node), we created two queues one for long jobs and the other for short jobs making sure to reduce interference between the jobs themselves - intra-disturbance - running on a same set of cores dynamically, and towards memcached - inter-disturbance - running on a different set of cores.

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 following table illustrates the SLO violation ratio for memcached across the three different runs:

Number of execution	SLO violation ratio %
Run 1	0.0%
Run 2	0.0%
Run 3	0.0%
Average	0.0%

job name	mean time [s]	std [s]
blackscholes	105.86	10.65
canneal	229.13	16.46
dedup	44.17	4.52
ferret	257.05	2.55
freqmine	445.74	7.58
radix	65.31	1.23
vips	103.64	5.12
total time	767.39	3.25

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: See Figure 7, Figure 8, Figure 9, Figure 10, Figure 11, Figure 12.

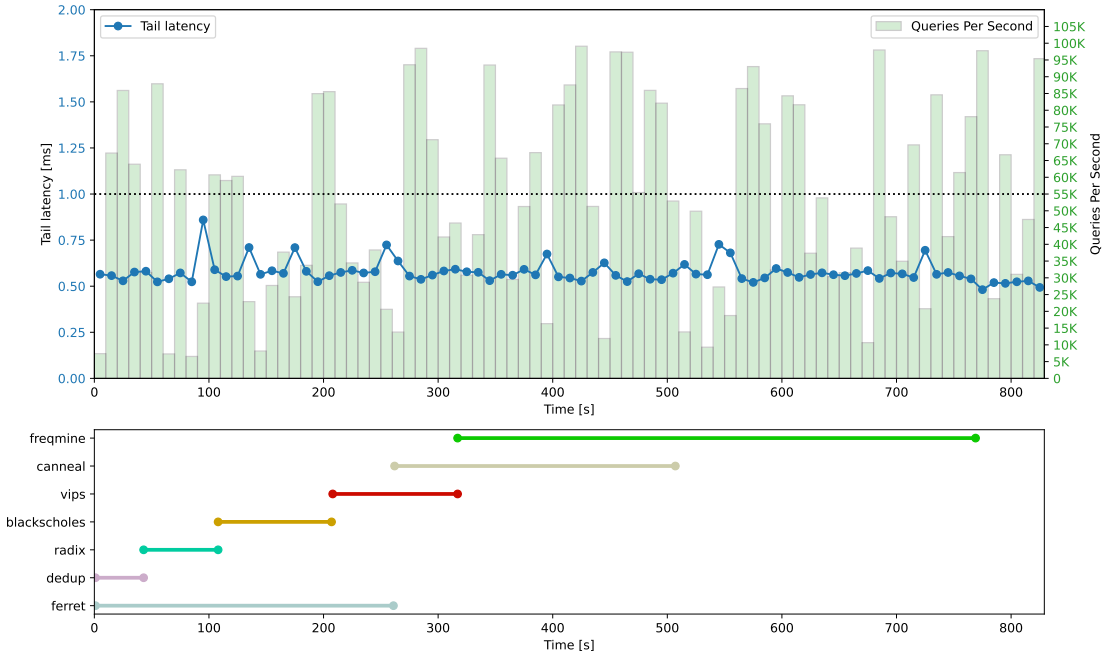

Figure 7: Plot 1A above shows 95th percentile latency and varying qps load over time for the first run. Plot 1A below shows PARSEC program runtime duration and scheduling across time which yielded the corresponding results in the upper plot.

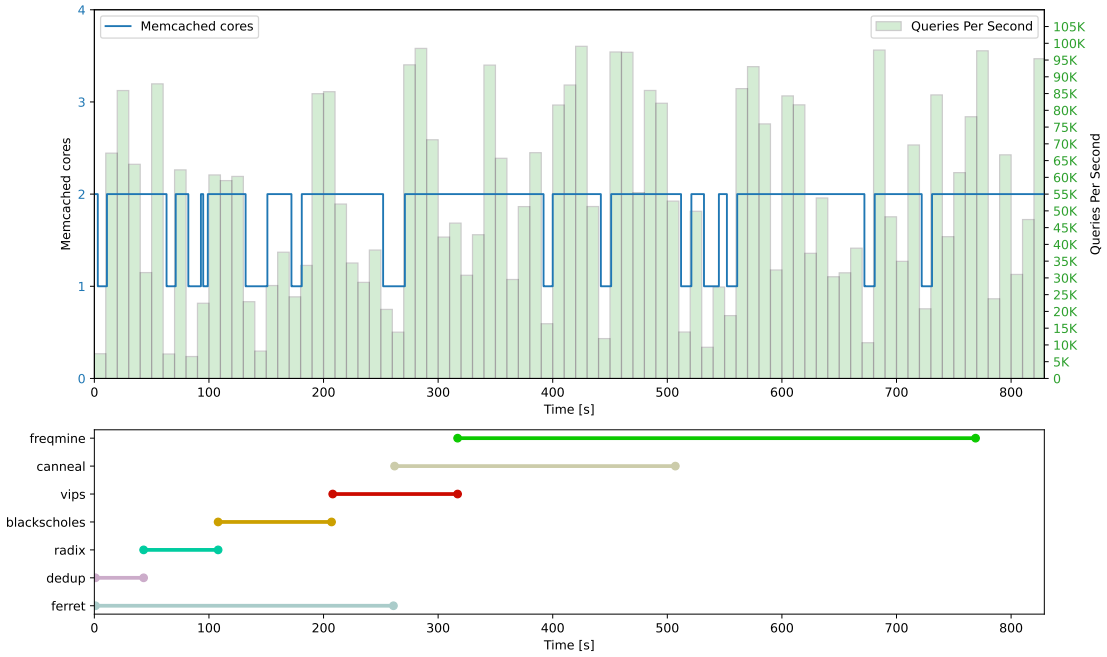

Figure 8: Plot 1B above shows number of cpu cores allocated to memcached and varying qps load over time for the first run. Plot 1B below shows PARSEC program runtime duration and scheduling across time which yielded the corresponding results in the upper plot.

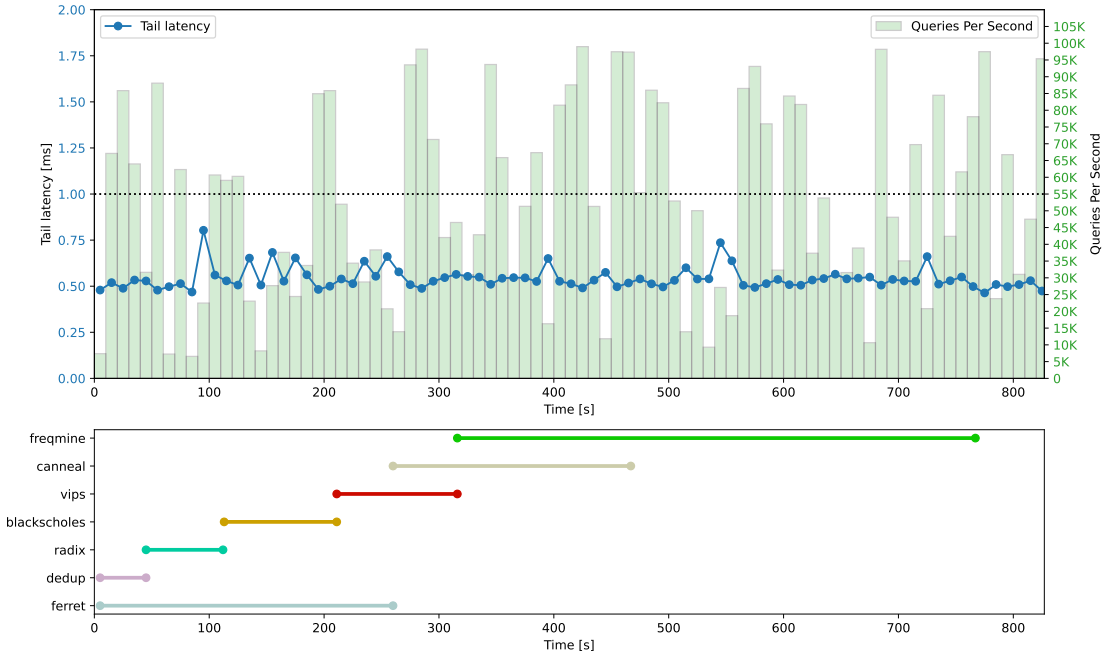

Figure 9: Plot 2A above shows 95th percentile latency and varying qps load over time for the second run. Plot 2A below shows PARSEC program runtime duration and scheduling across time which yielded the corresponding results in the upper plot.

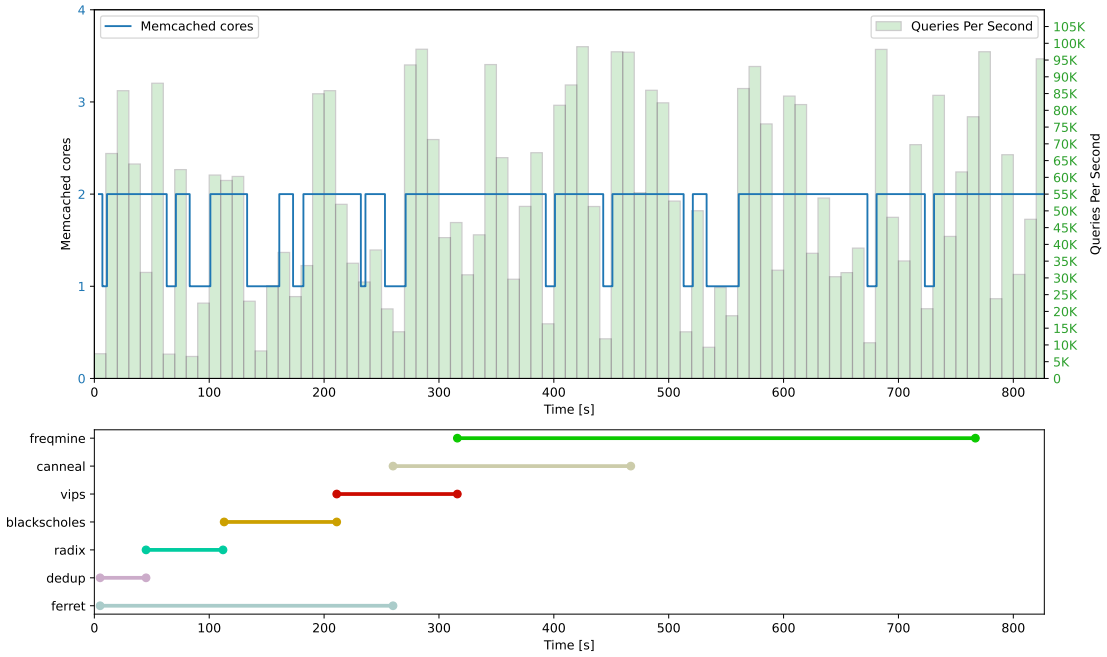

Figure 10: Plot 2B above shows number of cpu cores allocated to memcached and varying qps load over time for the second run. Plot 2B below shows PARSEC program runtime duration and scheduling across time which yielded the corresponding results in the upper plot.

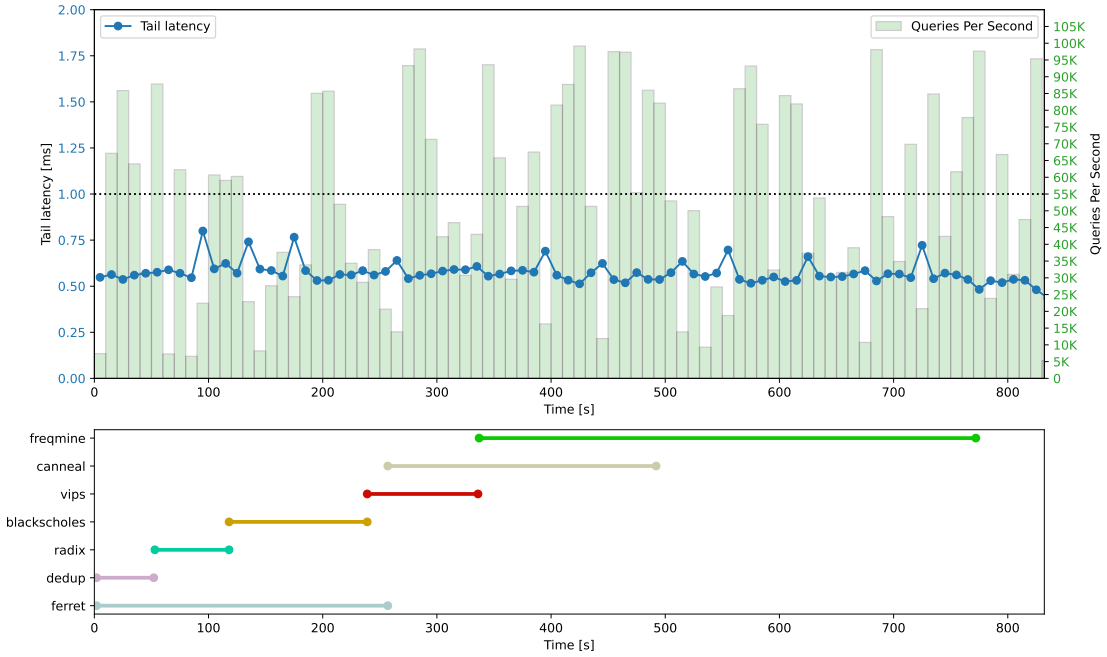

Figure 11: Plot 3A above shows 95th percentile latency and varying qps load over time for the third run. Plot 3A below shows PARSEC program runtime duration and scheduling across time which yielded the corresponding results in the upper plot.

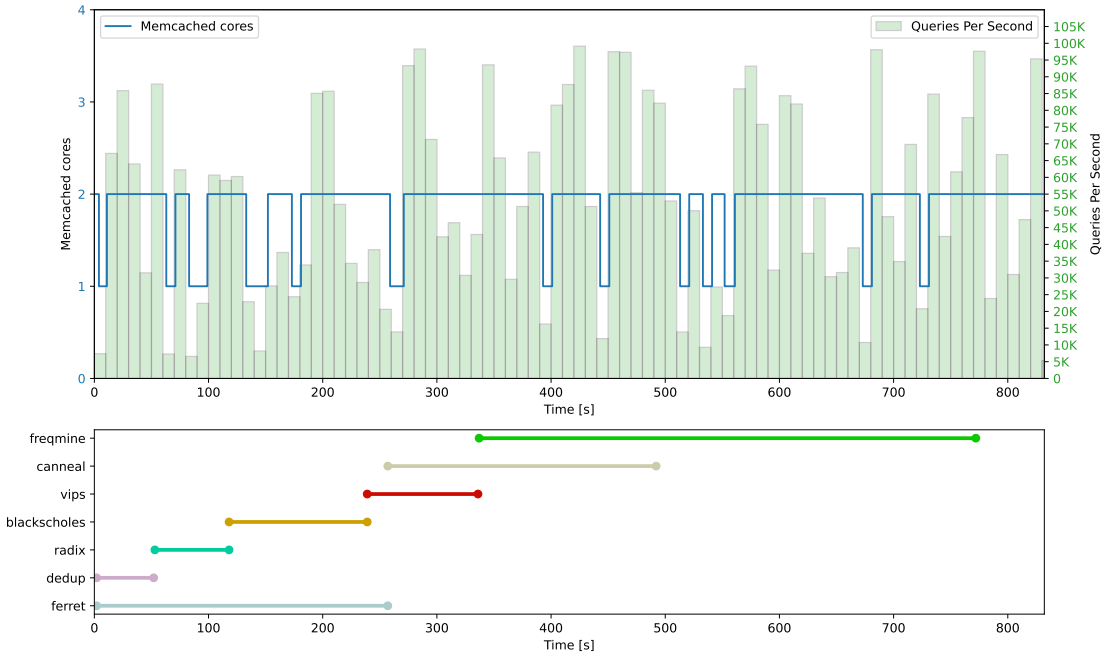

Figure 12: Plot 3B above shows number of cpu cores allocated to memcached and varying qps load over time for the third run. Plot 3B below shows PARSEC program runtime duration and scheduling across time which yielded the corresponding results in the upper plot.

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

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

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

Summary: When the parameter `qps_interval` is set to 5, the total running time of the jobs becomes slightly higher since memcached runs more time on 2 cores compared to when `qps_interval` is 10, due to the load changing more often and our switching cores policy. Moreover, the tail latency is generally less stable and reaches higher values.

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 ran the modified `mcperf` load trace with 5 second time interval in 3 different executions, in which we got the following SLO violation ratios for memcached:

Number of execution	SLO violation ratio %
Run 1	0.0%
Run 2	0.0%
Run 3	0.0%
Average	0.0%

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 checked the memcached SLO violation ratio across different `qps_interval` values from $10s$ (as in Part 4.3) to $1s$. In the case of $1s$ the obtained SLO violation ratios are the following:

Number of execution	SLO violation ratio %
Run 1	0.006%
Run 2	0.004%
Run 3	0.004%
Average	0.005%

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

Answer: We chose $1s$ as the smallest `qps_interval` in our runs. If we had chosen a `qps_interval` smaller than $1s$, we would be only analyzing fractions of QPS. In addition, looking at the mcperf documentation, the latter ensures a load in queries per second which might not be uniformly distributed over that second (the client agents could be non uniform, farther away or other factors). Thus, all we can assume is a steady load of queries per `second`, and not on fractions of a second. Moreover, our controller contains a call to sleep function for $0.25s$: choosing a `qps_interval` lower than this value would lead to skipping CPU utilization checks over some intervals.
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: See Figure 13, Figure 14, Figure 15, Figure 16, Figure 17 and Figure 18.

job name	mean time [s]	std [s]
blackscholes	169.25	2.12
canneal	280.46	3.15
dedup	49.14	4.46
ferret	368.85	6.04
freqmine	519.74	7.78
radix	73.88	9.12
vips	118.97	8.49
total time	934.28	2.48

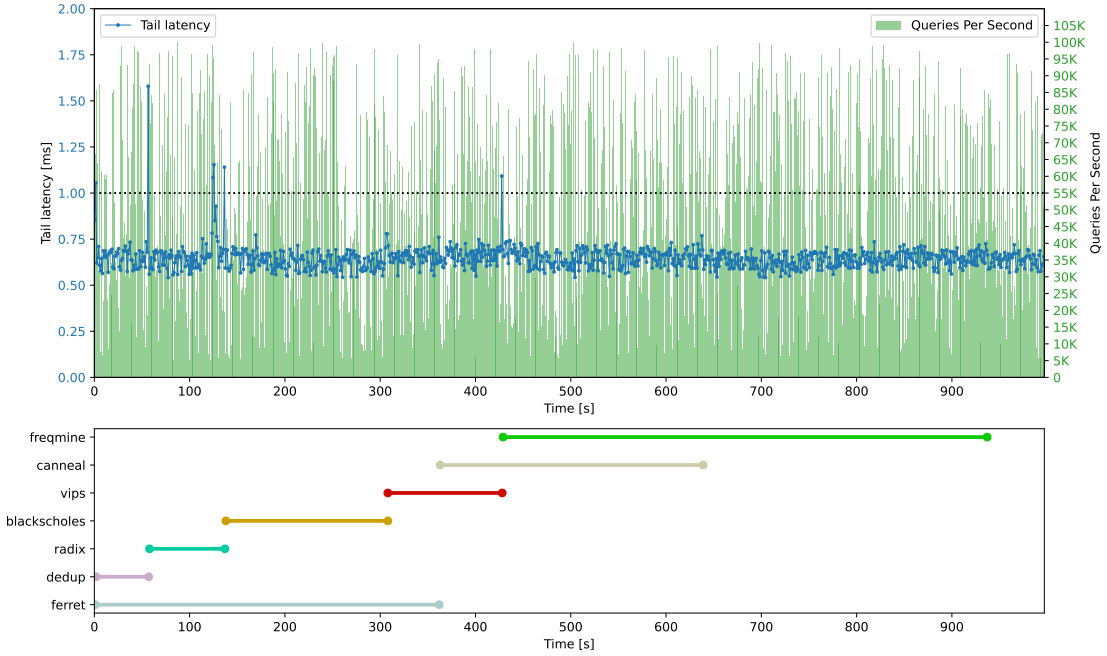

Figure 13: Plot 1A above shows 95th percentile latency and varying qps load over time for the first run. Plot 1A below shows PARSEC program runtime duration and scheduling across time which yielded the corresponding results in the upper plot.

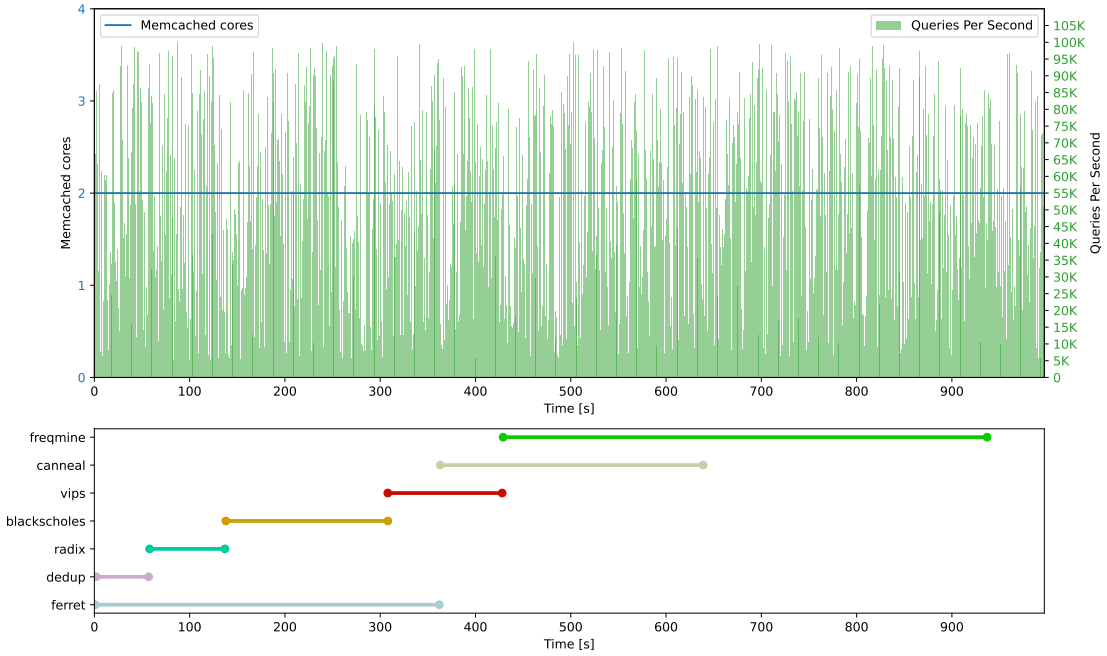

Figure 14: Plot 1B above shows number of cpu cores allocated to memcached and varying qps load over time for the first run. Plot 1B below shows PARSEC program runtime duration and scheduling across time which yielded the corresponding results in the upper plot.

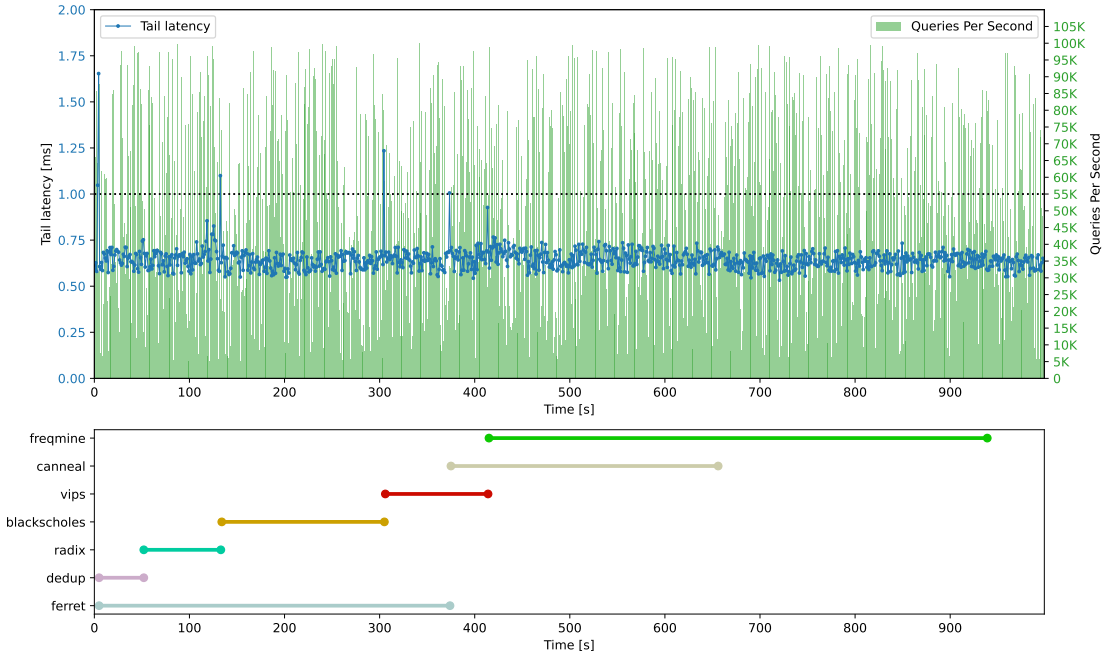

Figure 15: Plot 2A above shows 95th percentile latency and varying qps load over time for the second run. Plot 2A below shows PARSEC program runtime duration and scheduling across time which yielded the corresponding results in the upper plot.

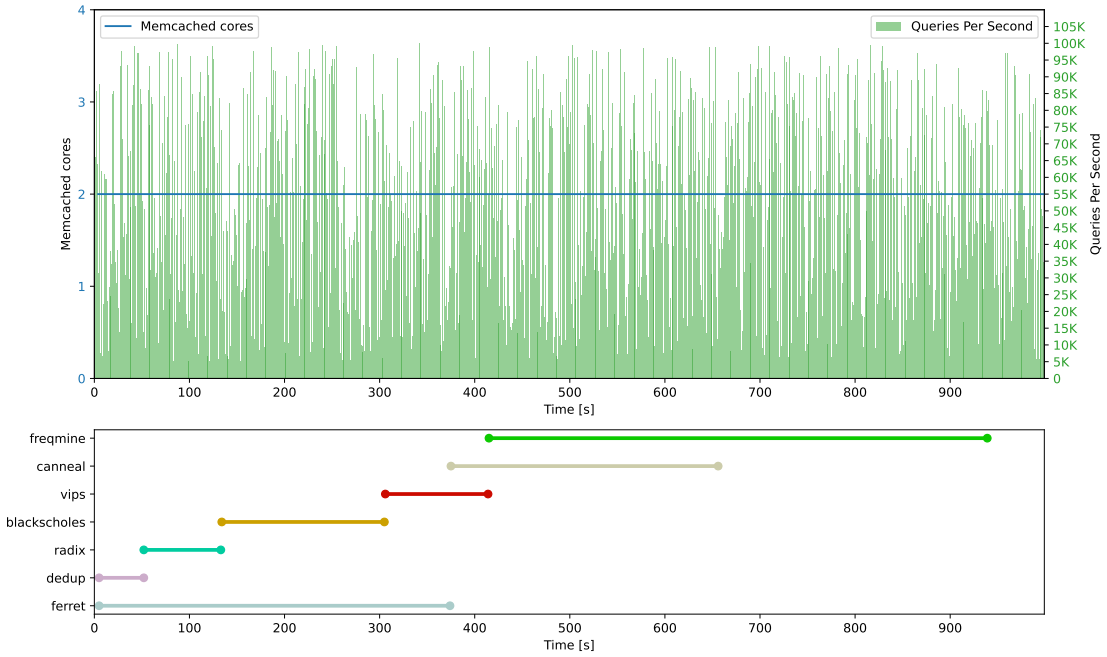

Figure 16: Plot 2B above shows number of cpu cores allocated to memcached and varying qps load over time for the second run Plot 2B below shows PARSEC program runtime duration and scheduling across time which yielded the corresponding results in the upper plot.

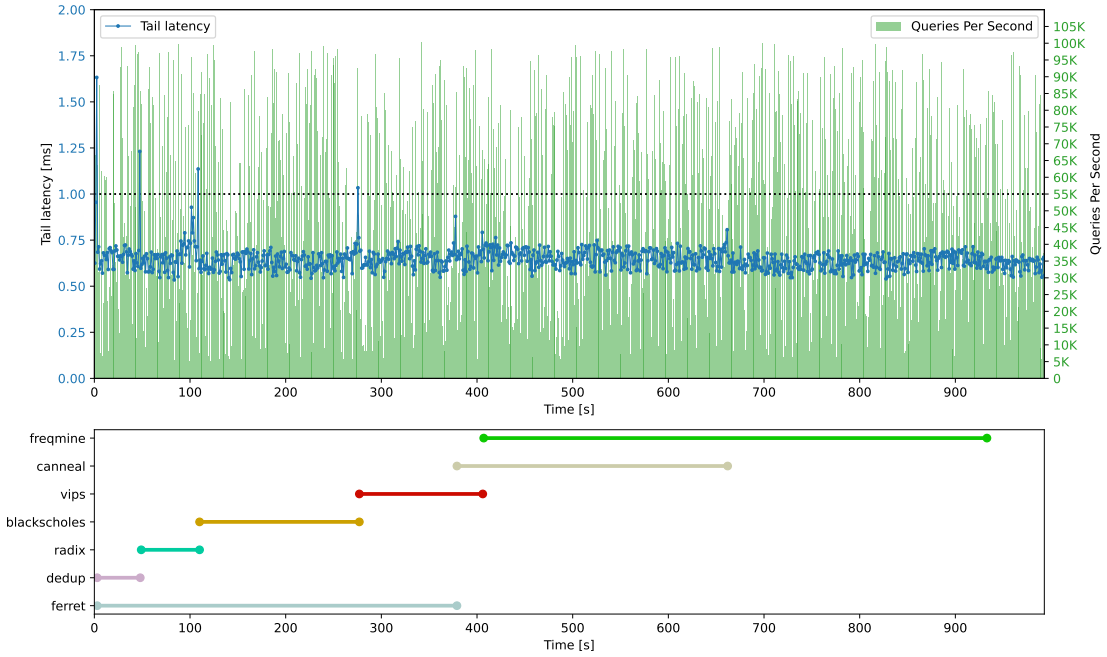

Figure 17: Plot 3A above shows 95th percentile latency and varying qps load over time for the third run. Plot 3A below shows PARSEC program runtime duration and scheduling across time which yielded the corresponding results in the upper plot.

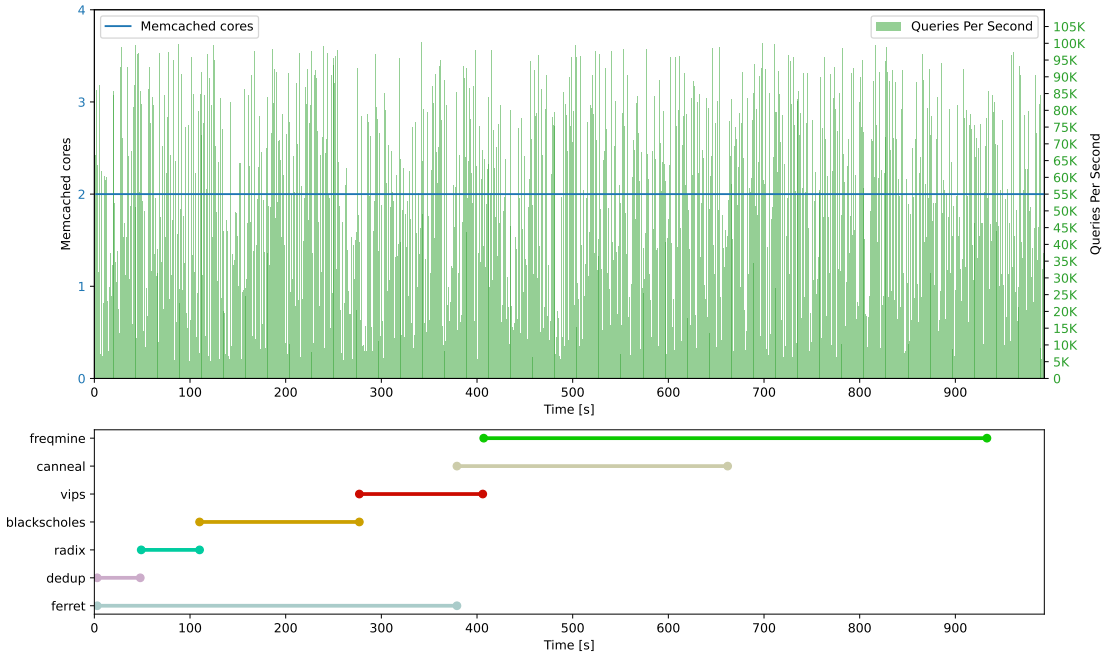

Figure 18: Plot 3B above shows number of cpu cores allocated to memcached and varying qps load over time for the third run Plot 3B below shows PARSEC program runtime duration and scheduling across time which yielded the corresponding results in the upper plot.