Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Cloud Computing Architecture

Semester project report

Group 49
Kehong Liu - 22943385
Yuening Yang - 21947502
Minjing Shi - 22943971

Systems Group
Department of Computer Science
ETH Zurich
July 20, 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.

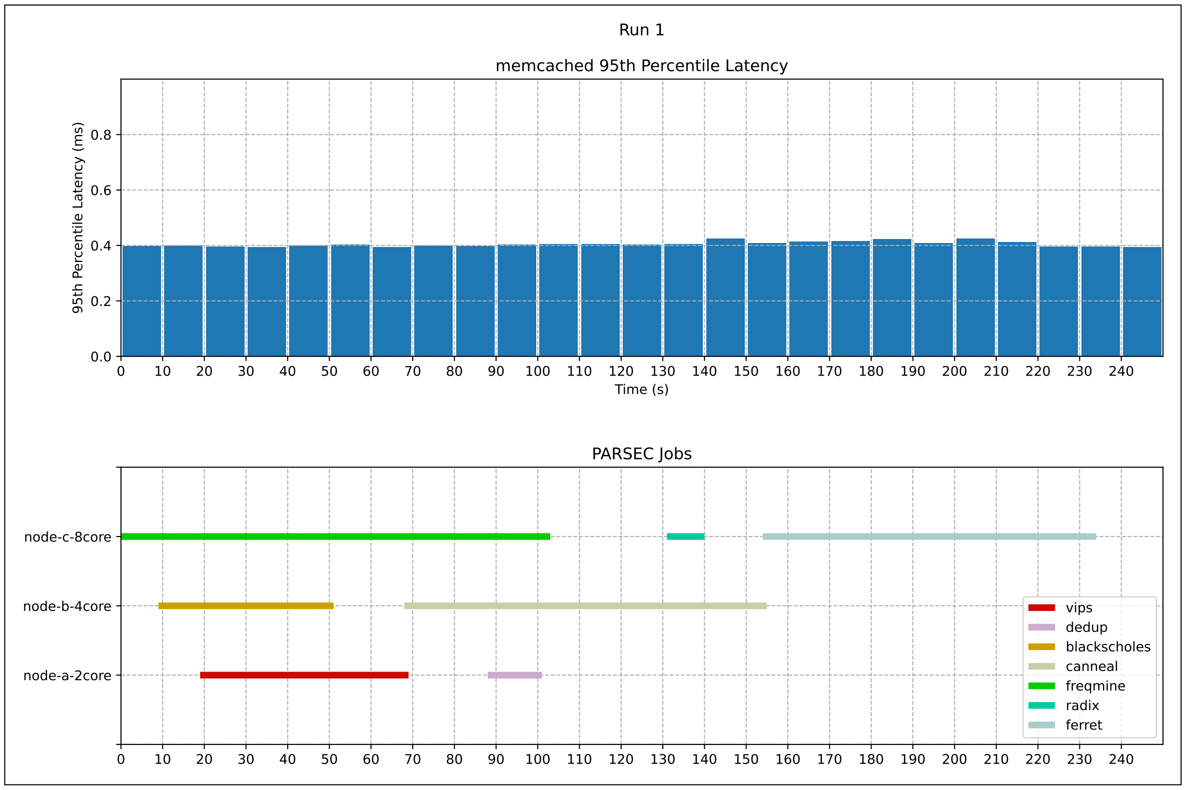
   Answer:
   The SLO violation ratio for memcached for all three runs is 0%, all 95th percentile latency is below 1ms.

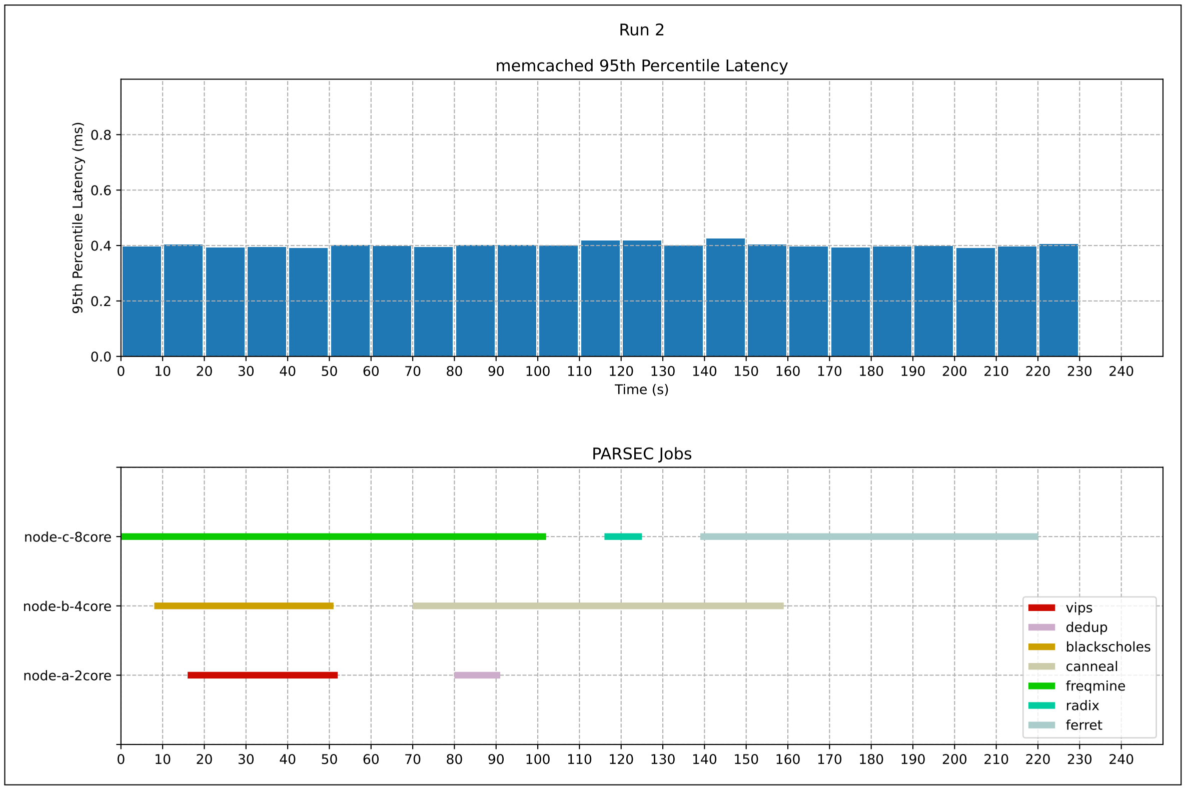
   Create 3 bar plots (one for each run) of memcached p95 latency (y-axis) over time (x-axis) with annotations showing when each PARSEC job started and ended, also indicating the machine they are running on. Using the augmented version of mcperf, you get two additional columns in the output: `ts_start` 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.

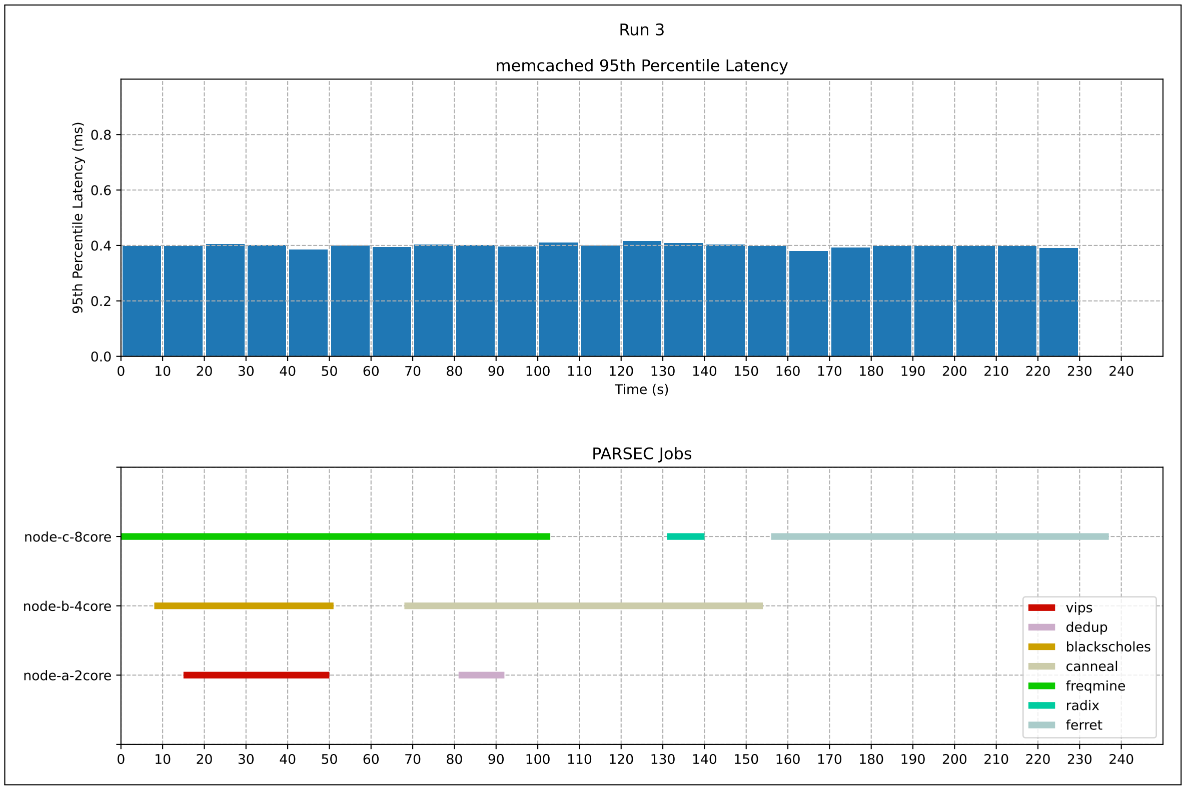
job name	mean time [s]	std [s]
blackscholes	42.67	0.47
canneal	87.33	1.25
dedup	11.67	0.94
ferret	80.67	0.47
freqmine	102.67	0.47
radix	9	0
vips	40.33	6.84
total time	230.33	7.41

   Plots:

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


Run 1
memcached 95th Percentile Latency
95th Percentile Latency (ms)
0.0
0.2
0.4
0.6
0.8
Time (s)
0 10 20 30 40 50 60 70 80 90 100 110 120 130 140 150 160 170 180 190 200 210 220 230 240
PARSEC Jobs
node-c-8core
node-b-4core
node-a-2core
vips
dedup
blackscholes
canneal
freqmine
radix
ferret


Run 2
memcached 95th Percentile Latency
95th Percentile Latency (ms)
0.0
0.2
0.4
0.6
0.8
Time (s)
0 10 20 30 40 50 60 70 80 90 100 110 120 130 140 150 160 170 180 190 200 210 220 230 240
PARSEC Jobs
node-c-8core
node-b-4core
node-a-2core
vips
dedup
blackscholes
canneal
freqmine
radix
ferret



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

- Which node does memcached run on? Why?
  Answer:
  memcached runs on `node-b-4core`. In Part 2 we concluded that memcached is best collocated with canneal or radix to meet the SLO since both of the jobs are not CPU or L1I intensive (which affects memcached performance the most) and are not heavy on shared resources such as memBW and LLC. Also from Part 2 we know that canneal has a noticeable speedup when going from 2 to 4 threads, but not much speedup on 8 threads, while radix speeds up a lot with 8 threads, which indicates radix should be on `node-c-8core` since it's the only machine that physically allows fore 8 threads. However, the other jobs that significantly speed up on 8 threads (and thus should be on `node-c-8core`)are ferret and freqmine, both are CPU and L1I intensive, and both require LLC. This indicates we cannot schedule memcached on `node-c-8core`, otherwise the CPU and L1I interferences can negatively affect memcached's performance, even if the CPU affinities are set to keep them running on different cores, there will still be large LLC interference, either slowing the jobs down or affect memcached's ability to meet the SLO. To minimize interference with the resource-intensive jobs on node-c-8core, canneal should be on `node-b-4core`, and this left us with the obvious choice of collocate memcached and canneal on `node-b-4core`. Note that blacksholes is also collocated with memcached on `node-b-4core`, since it has the most speedup when increasing thread from 2 to 4 but not much speedup on 8 threads like canneal. Although it is more CPU and L1I intensive, it's far from ferret and freqmine, and the issue can be mitigated by setting CPU affinity using `taskset`.

- Which node does each of the 7 PARSEC jobs run on? Why?
  Answer:
  - blackscholes: node-b-4core, as experimented in Part 2, blackscholes speeds up noticeably when jumping from 2 to 4 threads but does not speed up much for more than 4 threads. We decide not to schedule it on node-c-8-core for 2 reasons: if collocated with other jobs, since freqmine and ferret on node-c-8core are resource intensive, there will be a slowdown to all the jobs on that machine; if it's run before/after all the jobs, the extra cores are wasted since blacksholes does not gain performance for having 8 threads. Therefore it's run on the 4-core machine to use 3 threads (one core is dedicated to memcached). It is collocated with memcached because it is not resource intensive thus minimizes the interference with memcached.
  - canneal: node-b-4core, as experimented in Part 2, canneal speeds up noticeably when jumping from 2 to 4 threads but does not speed up much for more than 4 threads. We decide not to schedule it on node-c-8-core for 2 reasons: if collocated with other jobs, since freqmine and ferret on node-c-8core are resource intensive, there will be a slowdown to all the jobs on that machine; if it's run before/after all the jobs, the extra cores are wasted since canneal does not gain performance for having 8 threads. Therefore it's run on the 4-core machine to use 3 threads (one core is dedicated to memcached). It is collocated with memcached because it is not resource intensive thus minimizes the interference with memcached.
  - dedup: node-a-2core, as experimented in Part 2, dedup's performance barely increases after 2 threads but has some speed up when increasing thread count from 1 to 2. It is resource intensive so we decide to not collocate it with memcached on node-b-4core. We decide not to schedule it on node-c-8-core for 2 reasons: if collocated with other jobs, since dedup is resource intensive and freqmine and ferret on node-c-8core are also resource intensive (particularly, they all require LLC resources, which cannot be mitigated by setting CPU affinity), there will be a slowdown to all the jobs on that machine; if it's run before/after all the jobs, the extra cores are wasted since dedup does not gain performance for having 8 threads. Therefore it's run on the 2-core machine to use all threads.
  - ferret: node-c-8core, as experimented in part 2, ferret's performance benefits noticeably from having 8 threads compare to having 4 or fewer threads, thus it's run on the 8-core machine so it can use all 8 threads.
  - freqmine: node-c-8core, as experimented in part 2, freqmine's performance benefits noticeably from having 8 threads compare to having 4 or fewer threads, thus it's run on the 8-core machine so it can use all 8 threads.
  - radix: node-c-8core, as experimented in part 2, radix's performance benefits noticeably from having 8 threads compare to having 4 or fewer threads, thus it's run on the 8-core machine so it can use all 8 threads.
  - vips: node-a-2core, as experimented in Part 2, there is a noticeable speedup when increase the thread count from 1 to 2. Although there's also some speedup from increasing thread count from 2 to 4, we decided not to collocate it with memcached on the node-b-4core since it's resource intensive. We also decide not to schedule it on node-c-8-core for 2 reasons: if collocated with other jobs, since freqmine and ferret on node-c-8core are also resource intensive (particularly, they all require LLC resources, which cannot be mitigated by setting CPU affinity), there will be a slowdown to all the jobs on that machine; if it's run before/after all the jobs, the extra

cores are wasted since vips does not gain performance for having 8 threads. Thus, we put it on the 2-core machine to use all threads.

- Which jobs run concurrently / are colocated? Why?
  Answer:
  The scheduling policy can be described as follows:
  - node-a-2core:
    - core 0-1: vips → dedup
  - node-b-4core:
    - core 0: memcached
    - core 1-3: blackscholes → canneal
  - node-c-8core:
    - core 0-7: freqmine → radix → ferret

  The jobs run sequentially (i.e. one job at a time) on each machine, only blackscholes and canneal are collocated with memcached. Having the jobs executed sequentially allows each job to take the most advantage of parallelism and minimize interference between jobs, thus minimizing the running time of each individual job. Blackshcoles and canneal are collocated with memcached because they do not require much CPU and L1I resources, which impacts memcached's performance the most. They also requires relatively few LLC and memBW, which also contributes to less interference with memcached. To further reduce the interference, `taskset` is used to force blacksholes and canneal run on 3 cores while core 0 is dedicated to memcached. This ensures memcached can meet its SLO, at the cost of blacksholes and cannel not being able to take advantage of having all 4 cores.

- In which order did you run 7 PARSEC jobs? Why?
  Order (to be sorted): at the same time: (blackscholes, freqmine, vips); canneal, dedup, radix, ferret
  Why: The choice of which job to run first on each machine is arbitrary, and the choice shouldn't matter since jobs on each machine are run sequentially (i.e. one job at a time). The jobs are run in this order because some jobs finish quicker so the next job can be scheduled on that machine. We chose to start blackscholes, freqmine, and vips are at the same time since they are on different machines. Blackscholes finishes first so canneal is run, then vips finishes so dedup is run. Then freqmine finishes, radix is run, and finally, ferret is run after radix finishes. As indicated by the plots, the Kubernetes scheduler caused some delay when starting blackscholes and vips (time the start command is issued vs time the jobs actually started)at the beginning, but the total time is not affected as jobs on the two nodes finished quicker, so we did not address it by issuing the start command earlier than freqmine. Each job after that also has some delay between the start command issued and actually starting. We also decided to not address the issue since the performance penalty is relatively small one node-c-8core and the delays do not contribute to the total run time on the other two nodes.

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

  - blackscholes: 3, Part 2 shows blacksholes has a noticeable speedup when increasing thread count from 2 to 4 but a less noticeable speedup from 4 to 8, so we scheduled it on node-b-4core. Since one core on that machine is dedicated to memcached, we used 3 threads for it.
  - canneal: 3, Part 2 shows canneal has a noticeable speedup when increasing thread count from 2 to 4 but a less noticeable speedup from 4 to 8, so we scheduled it on node-b-4core. Since one core on that machine is dedicated to memcached, we used 3 threads for it.
  - dedup: 2, Part 2 shows dedup has a noticeable speedup when increasing thread count from 1 to 2 and almost no speedup for more than 4 threads, so we used 2 threads for it.
  - ferret: 8, Part 2 shows ferret speeds up significantly when increasing the thread count from 4 to 8, so we used 8 threads for it.
  - freqmine: 8, Part 2 shows freqmine speeds up significantly when increasing the thread count from 4 to 8, so we used 8 threads for it.
  - radix: 8, Part 2 shows radix speeds up significantly when increasing the thread count from 4 to 8, so we used 8 threads for it.
  - vips: 2, Part 2 shows vips has a noticeable speedup when increasing thread count from 1 to 2. Although it has some speedup from increasing the thread count beyond 2, it is scheduled on node-a-2core as explained above, which physically limits the max number of threads it can use to 2. So we used 2 threads for it.

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

  Answer:
  The yaml files for launching PARSEC jobs and the memcached are modified. For the memached file, the `nodeSelector` field is modified so it runs on `node-b-4core`, and `taskset -c 0` is added to the command so it always runs on core 0. For the PARSEC files, `nodeSelector` field is modified so they run on the machines as specified above, and `-n` argument is modified in the command launching the jobs to allow the jobs to use the number of threads specified in the previous question. `taskset -c 1-3` is added to the command launching blackscholes and canneal to reduce interference with memcached since they are collocated on the same machine. The Kubernetes feature we used is `nodeSelector`, which allows us to specify which machine the pod should be run on.

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

  Answer:
  The idea is to schedule PARSEC jobs on machines that have enough cores to allow the jobs use to get the most significant speedup by multithreading, and evenly distribute the jobs across the machines. The first design choice we made is for each machine, we have jobs run one at a time with all available threads to minimize the time of each job. We chose not to have them run concurrently and use part of threads (share LLC and memBW with other jobs), or concurrently with all threads (also share CPU, L1I, L1D with other jobs). This choice is made because Part 2's experiment showed that some jobs speed up significantly by increasing the thread count, so we want to give it as many

threads as it needs. We decide not to collocate jobs to minimize the interference between them to further reduce the running time of each individual job. As mentioned before, if we have two jobs running concurrently on different cores, each of them cannot take full advantage of parallelism and there will still be LLC and memBW interference, and this can double the running time for some jobs. If we have two jobs running concurrently on all the cores there's some speedup from better parallelism, but there will be additional interference from CPU, L1I, L1D, which also has a great negative impact on each job's performance. We believe the total time to run faster jobs one at a time should still be faster than letting fewer jobs run slowly and waiting for the slowest one to finish.
Another design choice is to dedicate one core to memcached. This is because Part 1 showed us memcached is very sensitive to CPU and L1I interference, thus by dedicating one core to it can minimize such interference and make sure it can achieve the SLO. The tradeoff of this choice being whatever jobs collocated with memcached cannot take full advantage of parallelism, which makes the execution time longer.

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:

See Fig.1

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:

From Fig.1 we can see that running 2 threads on 2 cores has the lowest latency, while running 2 threads on 1 core is the worst case considering. It would be better to have more or an equal number of cores than threads to avoid several threads competing for CPU resources. There is a significant increase in latency on QPS=75000, and a single thread or a single core may not be enough for the workloads.

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:

Only the configuration of 2 threads on 2 cores works without violating the 1ms latency SLO among those 4 cases. Therefore, we need $T = 2$ threads and $C = 2$ cores.

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:

Since both 1 thread on 1 core and 1 thread on 2 cores cannot guarantee the 1ms 95th percentile latency SLO, the number of threads must be 2.

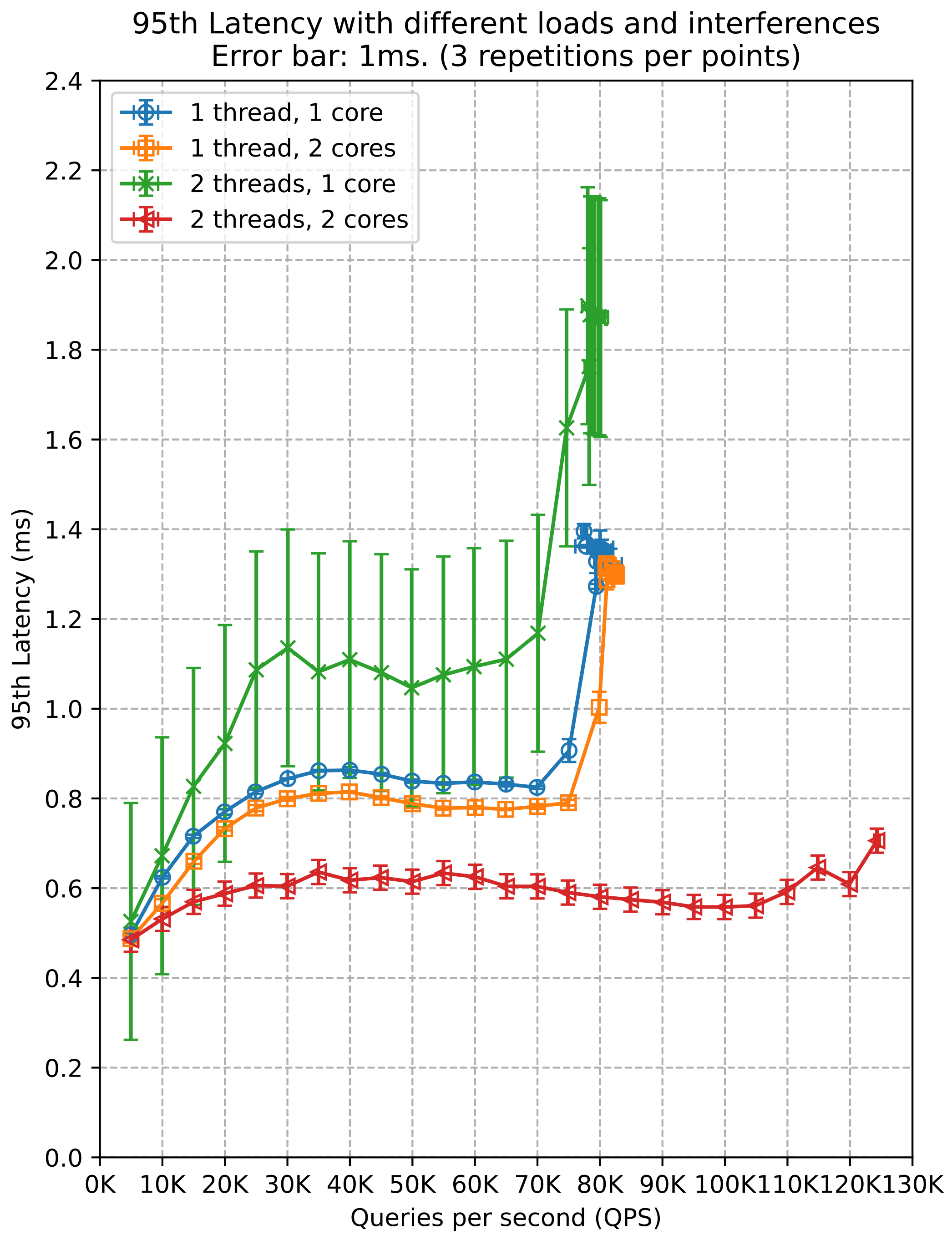
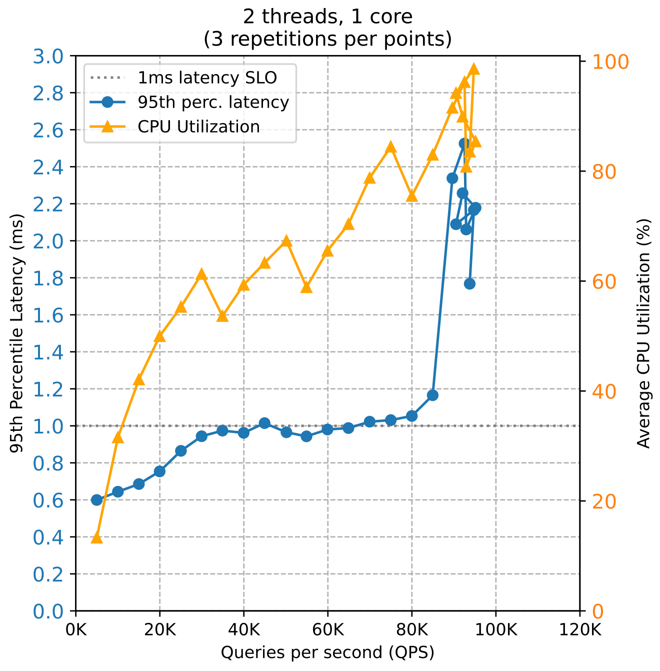


Figure 1: Memcached performs best with two threads on two cores while worst with two threads on a single core



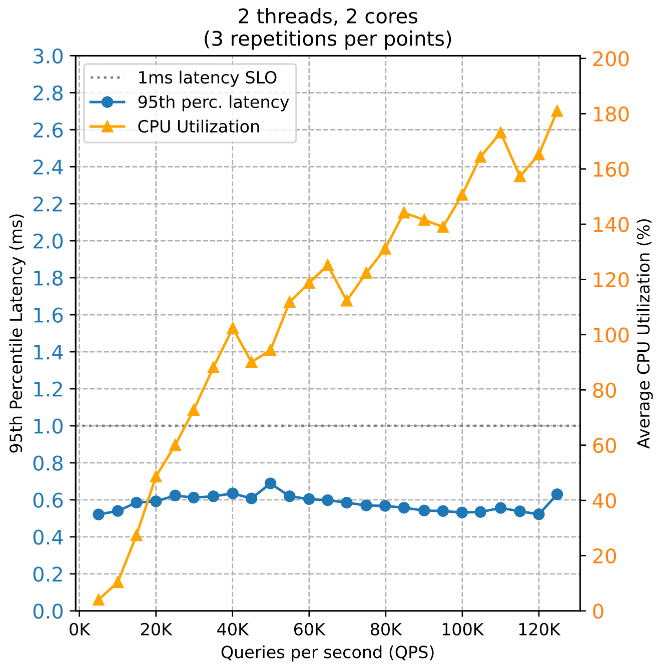


Figure 2: CPU utilization and 95th latency change under different QPS with 2 threads on 1 or 2 core(s) respectively.

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 Fig.2

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:
  We schedule all jobs sequentially except dedup. We will start vips concurrently with memcached and dedup, with dedup occupying core 1, and vips occupying core 2 and 3. The jobs are split into two queues, where the first queue include only dedup, and the second queue include the rest, for which we call the main queue. Once dedup is done on queue 1, we will dynamically allocate resource to the job running on queue 2 (main queue), according to the utilization of CPU from memcached. items on the second queue are also executed according to a specific order.
- How do you decide how many cores to dynamically assign to memcached? Why?
  Answer:
  We found that memcached is stable enough when running on 2 cores and 2 threads, hence we will set the affinity of memcached to core 0,1, and fix memcached to run on 2 threads. Other than that, we do not manually interfere with setting the CPU utilization of memcached. We believe that starting, pausing, or changing the number of assigned cores to the jobs is enough to handle any minor changes to memcached's cpu utilization.
- How do you decide how many cores to assign each PARSEC job? Why?
  Answer:
  - blackscholes: 2-3
  - canneal: 2-3
  - dedup: 1
  - ferret: 2-3
  - freqmine: 2-3
  - radix: 2
  - vips: 2-3

  We found that dedup can finish in reasonable amount of time with just one core and one thread, while radix can also run quite fast with just 2 cores. All other jobs may benefit to varying degrees in running time if they are given more cores, according to our observation on core allocation. If memcached does not consume resources consisting of more than one core, we would try to let these jobs run in as many cores as possible.

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

  Answer:

  - blackscholes: 4
  - canneal: 4
  - dedup: 2
  - ferret: 8
  - freqmine: 8
  - radix: 8
  - vips: 2

  According to our observations from part 1 and 2 of the project, dedup can finish in a reasonable amount of time with just one thread and have a limited speedup with 2 threads, and it does not benefit much from more cores and threads. The same is for vips. canneal and blackscholes are speed up significantly from 1 thread to 4 threads and therefore assigned 4 threads. radix, ferret, and freqmine are assigned 8 threads as they significantly speedup until 8 threads.

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

  Answer:

  At the starting point, memcached, dedup, and vips are all running. Dedup occupies core 1, and memcached occupies core 0 and 1, hence these two are collocated on core 1. However, dedup will not run until the utilization of memcached goes below 50 percent on core 0. vips occupy cores 2 and 3, as with all other jobs on queue 2 when they start running. These jobs will be temporarily assigned an extra core (core 1) when memcached does not utilize more than one core, and while dedup as finished executing. We believe that letting memcached and other jobs run on separate cores and threads will greatly help increase the stability of memcached.

  To ensure memcached's latency is stable enough, we would only assign the extra core to jobs on the main queue when memcached's core utilization goes under 50 percent on core 0. In this way, we can leave some headroom for memcached to scale up without incurring large penalties on latencies.

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

  Order (to be sorted):

  queue 1: dedup

  queue 2: vips, radix, blackscholes, canneal, freqmine, ferret

  Why: Our main design goal is to ensure that *memcached*'s SLO violation ratio is as small as possible. Since *memcached*'s 95th percentile latency is guaranteed with 2 threads and 2 cores, we can dynamically scale it up to 2 cores or down to 1 core.

  We have thus 2 or 3 cores left for running the jobs and those are very limited resources. Therefore, we tend to schedule jobs sequentially to avoid them interfering with each other.

An exception is dedup, which runs very fast and hard to speed up. We schedule it to be run in parallel with vips which is also hard to speed up by adding more threads and has a short running time. It can also potentially be scheduled with radix. This is because although radix speeds up significantly with more cores and threads, the real amount of time reduced is quite limited as the job itself runs very fast. In addition, it can only be run with 2^x number of threads, which also give a chance for dedup to run parallel with it.

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

  Answer:

  The first difference is the level of parallelism. In part 3, jobs are run on 3 separate machines, thus interference between different jobs are minimal compared with part 4, where all jobs have to be executed on a single machine. In part 3, there is relatively high level of parallelization of jobs, with up to 4 jobs (including memcached) running concurrently. In part 4, we ensured there is less parallelization involved between different jobs, with dedup being an exception as it does not consume much resource when running. Hence, jobs are mostly run sequentially, with dedup occasionally running concurrently with the main job queue.

  Another difference is with resource allocation of jobs. For part 3, almost all jobs are running at their peak performance, since there are enough machines to allocate resources to different jobs. Time and resource-consuming jobs such as freqmine and ferret are given 8 cores, while less resource consuming jobs such as dedup are given 2 cores. For part 4, all jobs are assigned a maximum of 3 cores, according to the utilization of memcached on the cpu. Only the maximum number of threads are allocated so that jobs can run as quickly as possible.

  Finally, there is the difference in execution order. In part 3, jobs are executed concurrently on three separate machines, without considering which job to execute first since each machine executes all jobs sequentially without interference. However, in part 4, we have to consider the impacts of interference on jobs with higher consumption of resources. Therefore, we choose to run jobs that consume less power (e.g. vips and radix), and executes the fastest at first, and we put those jobs which require more power (e.g. freqmine and ferret) towards the end of the queue.

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

  Answer:

  We use taskset command to set the cpu affinity of memcached to core 0 and 1, hence memcached can only occupy these two cores. During the execution of jobs, we dynamically adjust the number of cores for each job using the docker update command cpu-set. To pause and unpause jobs, we use the docker command on python. We first check if the job is valid and running by reloading the container, then we call the respective commands to pause and unpause the jobs.

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

Answer:

We use subprocess on python to monitor the core usage of memcached. We keep monitoring the usage with a loop until all jobs finish executing. For every 0.4 seconds, the usage of memcached is refreshed, and the according updates to jobs are done based on our policy written before. Additionally, we keep an optional COOL_TIME parameter, which prevents the docker containers to update too frequently, since we found in earlier experiments that containers are likely to crash if we update them too frequently (e.g. every 0.1 seconds).

3. [**23 points**] Run the following `mcperf` memcached dynamic load trace:

```
$ ./mcperf -s INTERNAL_MEMCACHED_IP --loadonly
$ ./mcperf -s INTERNAL_MEMCACHED_IP -a INTERNAL_AGENT_IP \
           --noload -T 16 -C 4 -D 4 -Q 1000 -c 4 -t 1800 \
           --qps_interval 10 --qps_min 5000 --qps_max 100000 \
           --qps_seed 3274
```

Measure memcached and PARSEC performance when using your scheduling policy to launch workloads and dynamically adjust container resource allocations. Run this workflow 3 separate times. For each run, measure the execution time of each PARSEC job, as well as the latency outputs of memcached. For each PARSEC application, compute the mean and standard deviation of the execution time across three runs. Compute the mean and standard deviation of the total time to complete all jobs. Fill in the table below. Also, compute the SLO violation ratio for memcached for each of the three runs; the number of data points with 95th percentile latency > 1ms, as a fraction of the total number of datapoints. You should only report the runtime, excluding time spans during which the container is paused.

Answer: The SLO violation ratio is all zero in three runs

job name	mean time [s]	std [s]
blackscholes	62.513	0.151
canneal	160.884	1.739
dedup	36.335	0.785
ferret	206.792	0.602
freqmine	347.463	0.688
radix	27.367	0.354
vips	59.825	1.057
total time	866.469	1.572

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

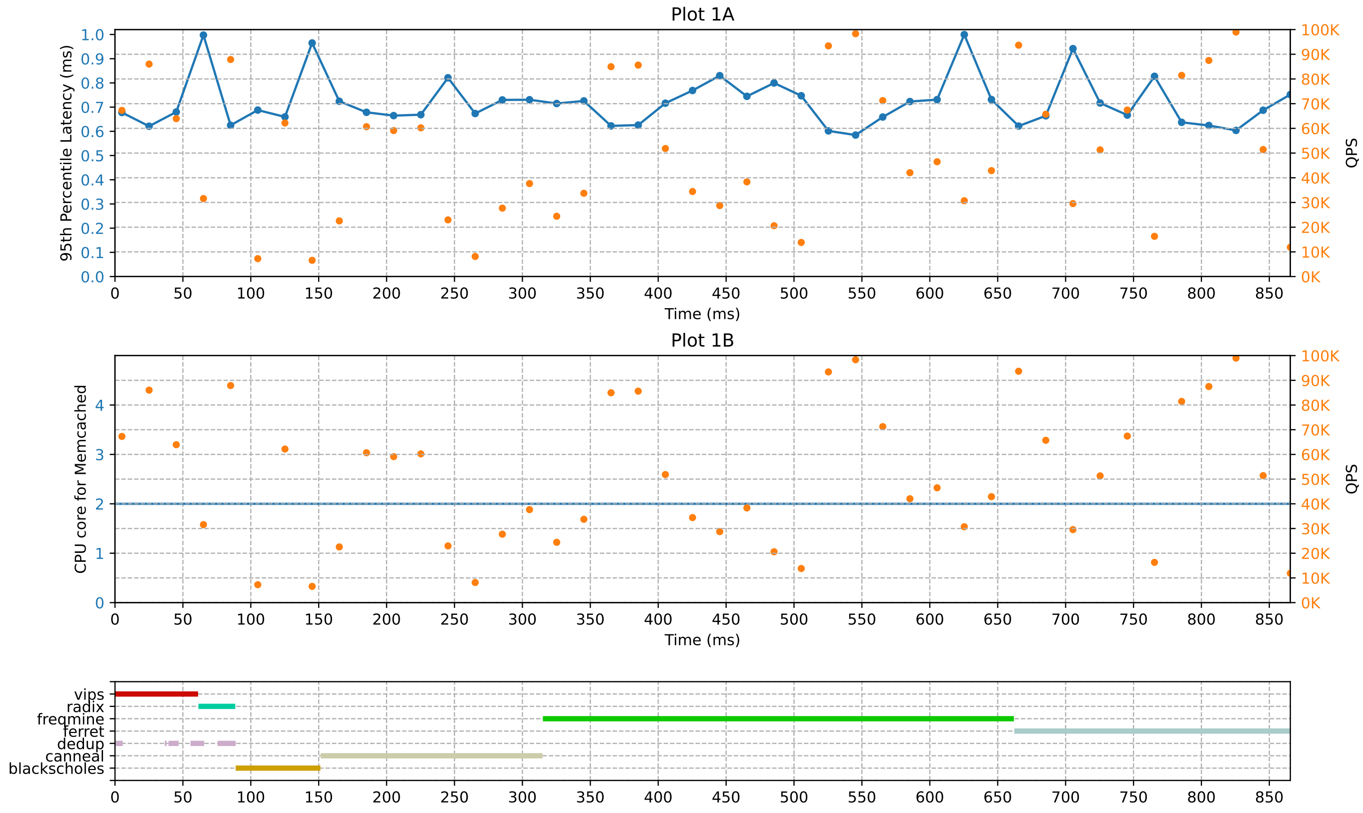


Figure 3: Run1: 95th percentile latency and CPU usage of *memcached* under dynamic load with frequency 10s. Together with the running time of jobs

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: Fig.3, Fig.4 and Fig.5 are the plots. Except for PlotA and PlotB, we also add an annotation plot to declare which job is running in which period.

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:

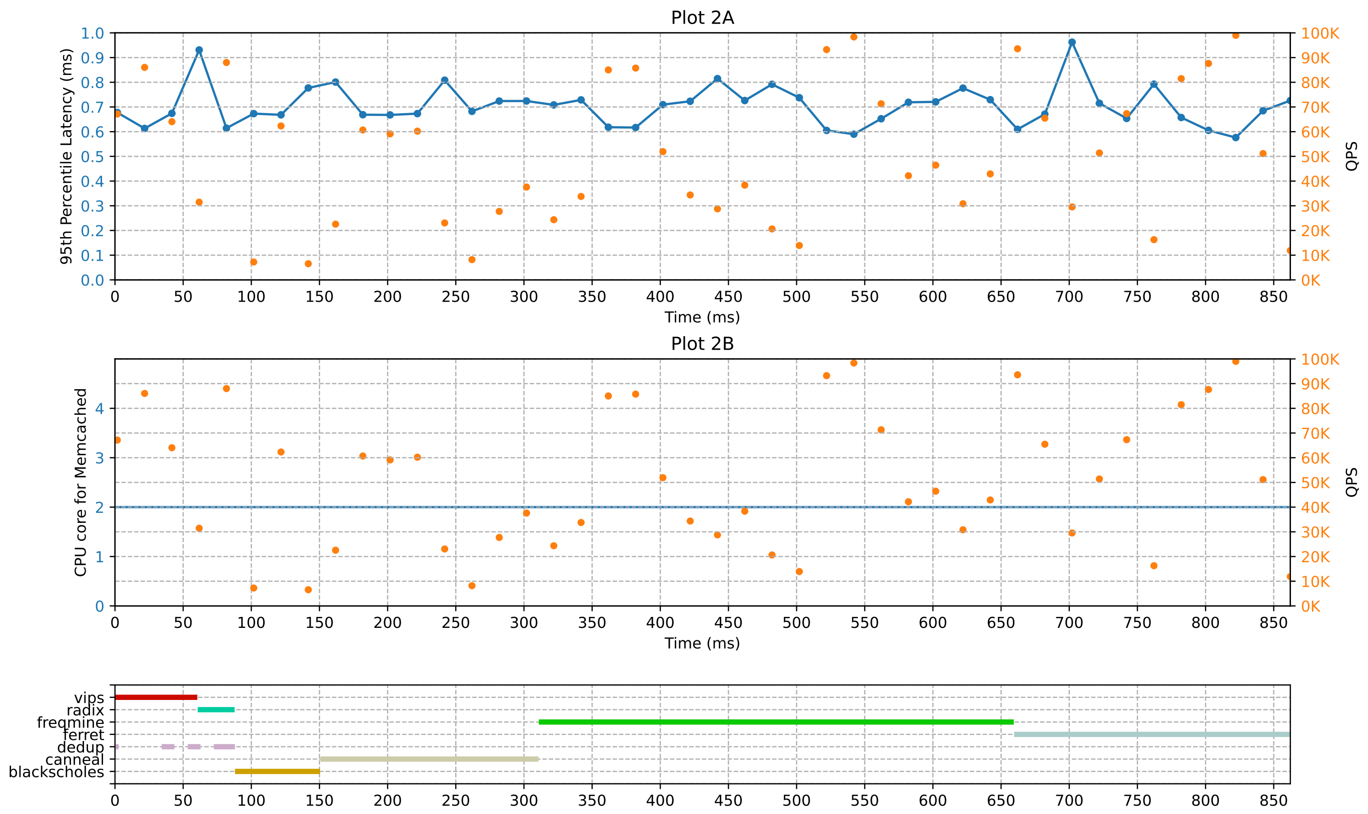


Figure 4: Run2: 95th percentile latency and CPU usage of *memcached* under dynamic load with frequency 10s. Together with the running time of jobs

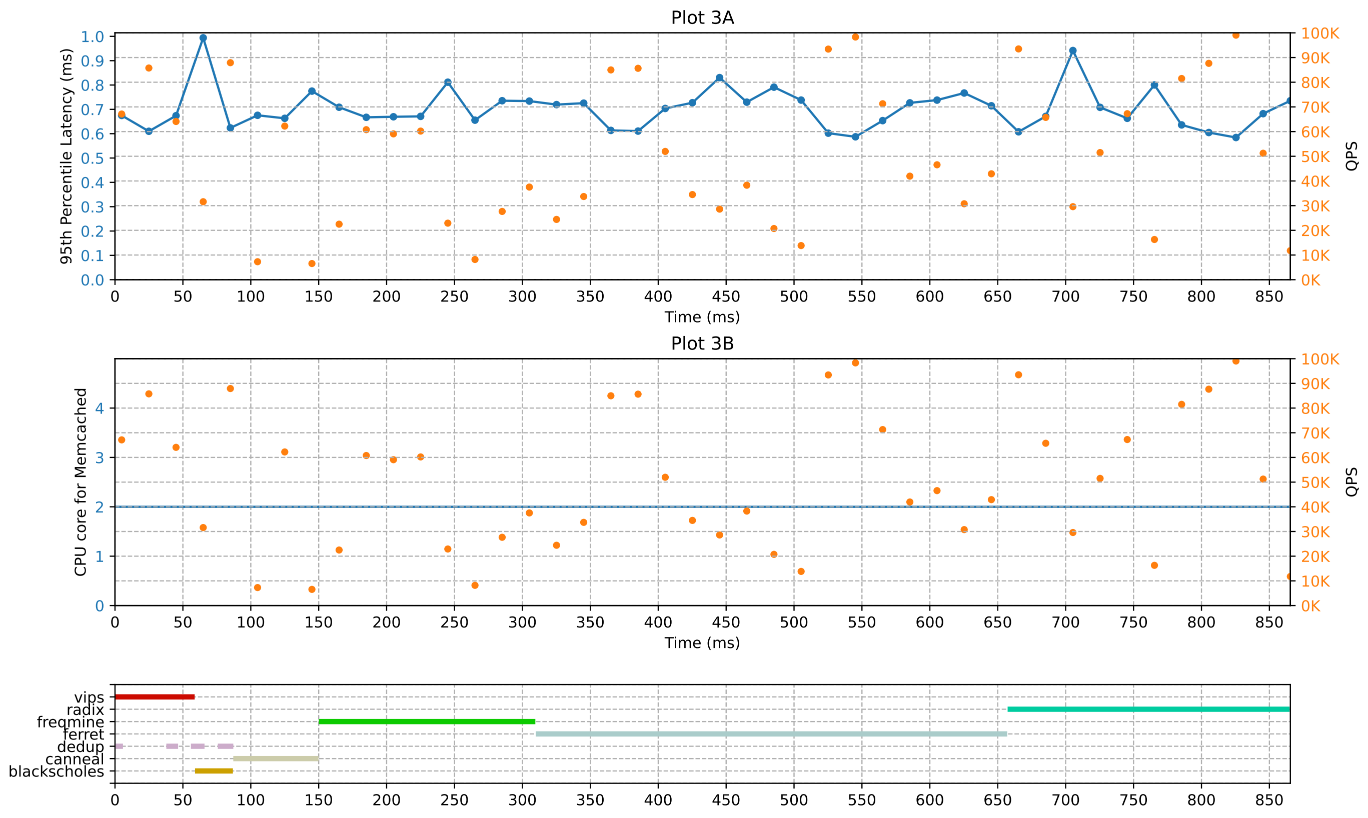


Figure 5: Run3: 95th percentile latency and CPU usage of *memcached* under dynamic load with frequency 10s. Together with the running time of jobs

By viewing the logs, we notice that since we launch/pause and adjust resources to jobs based on memcached CPU utilization, a higher load variability will significantly increase the frequency of the controller pausing/unpausing and changing the CPU assigned to jobs. In addition, most SLO violations happen at the beginning where dedup is running in parallel with another job.

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 SLO violation ratio for memcached in the three runs are 0%, 0.1%, 0.1%, respectively

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` is 1s

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

Answer:

Two features might be important.

First, we set a cooling period every time when memcached scales up, i.e. the scheduler reduce CPU allocated to jobs or pause a job. This means jobs cannot be unpaused or scale up inside this period. Making this period slightly longer will make the smallest `qps_interval` smaller

Second, the scheduler split the whole process into two part: dedup is running and dedup is not running. Since most SLO violations happen in the first part, we could give more resources to Memcached, i.e. only scale up jobs or unpause dedup when CPU utilization drops below a small number. A further option is to make this number smaller (now 50%) to make the smallest `qps_interval` smaller, however, in trade of the total running time.

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.

The SLO violation ratios for those three runs are 0.5%, 1.8%, 2.4%, respectively

Plots: Fig.6, Fig.7 and Fig.8 show the results

job name	mean time [s]	std [s]
blackscholes	74.081	0.533
canneal	169.008	1.236
dedup	36.050	0.553
ferret	225.490	3.690
freqmine	374.322	4.675
radix	26.520	0.282
vips	55.602	0.733
total time	926.746	7.069

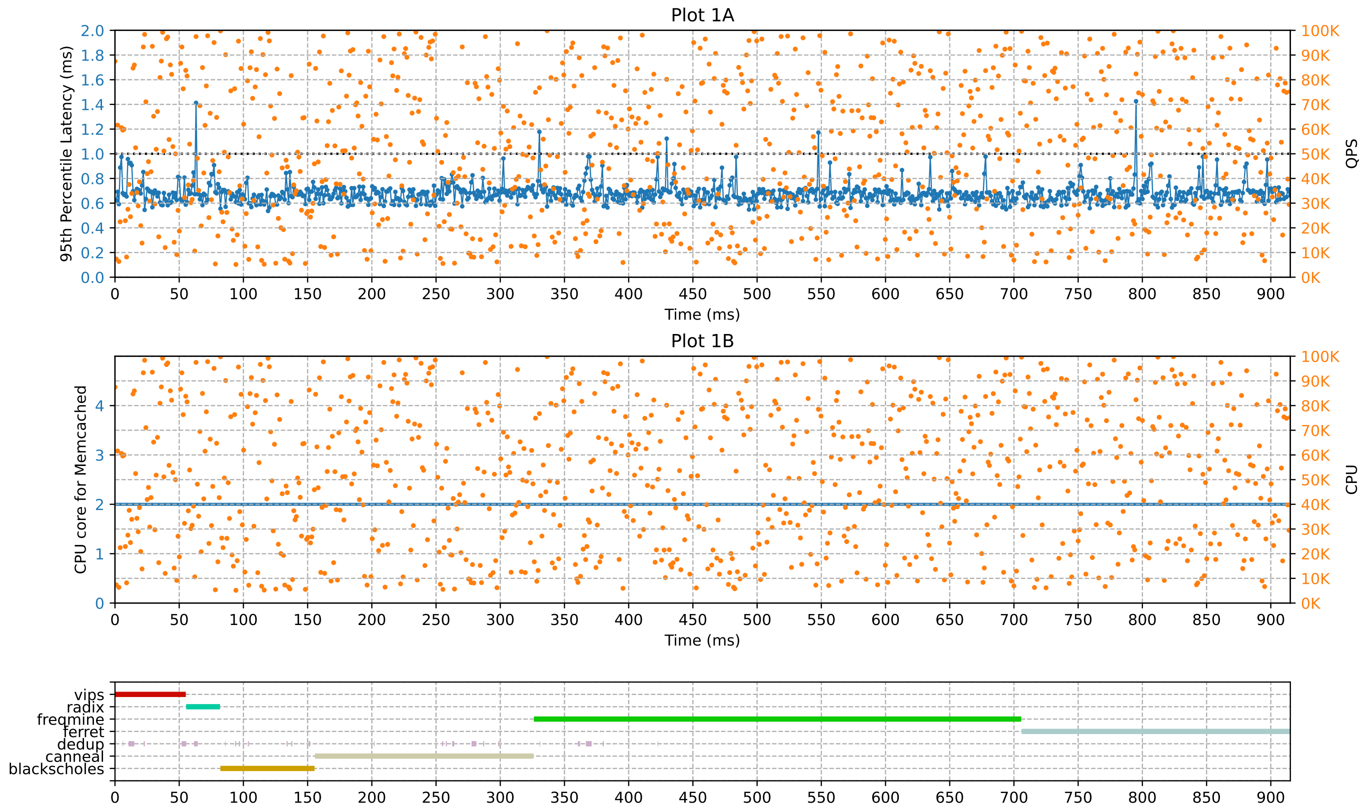


Figure 6: Run1: 95th percentile latency and CPU usage of *memcached* under dynamic load with frequency 1s. Together with the running time of jobs

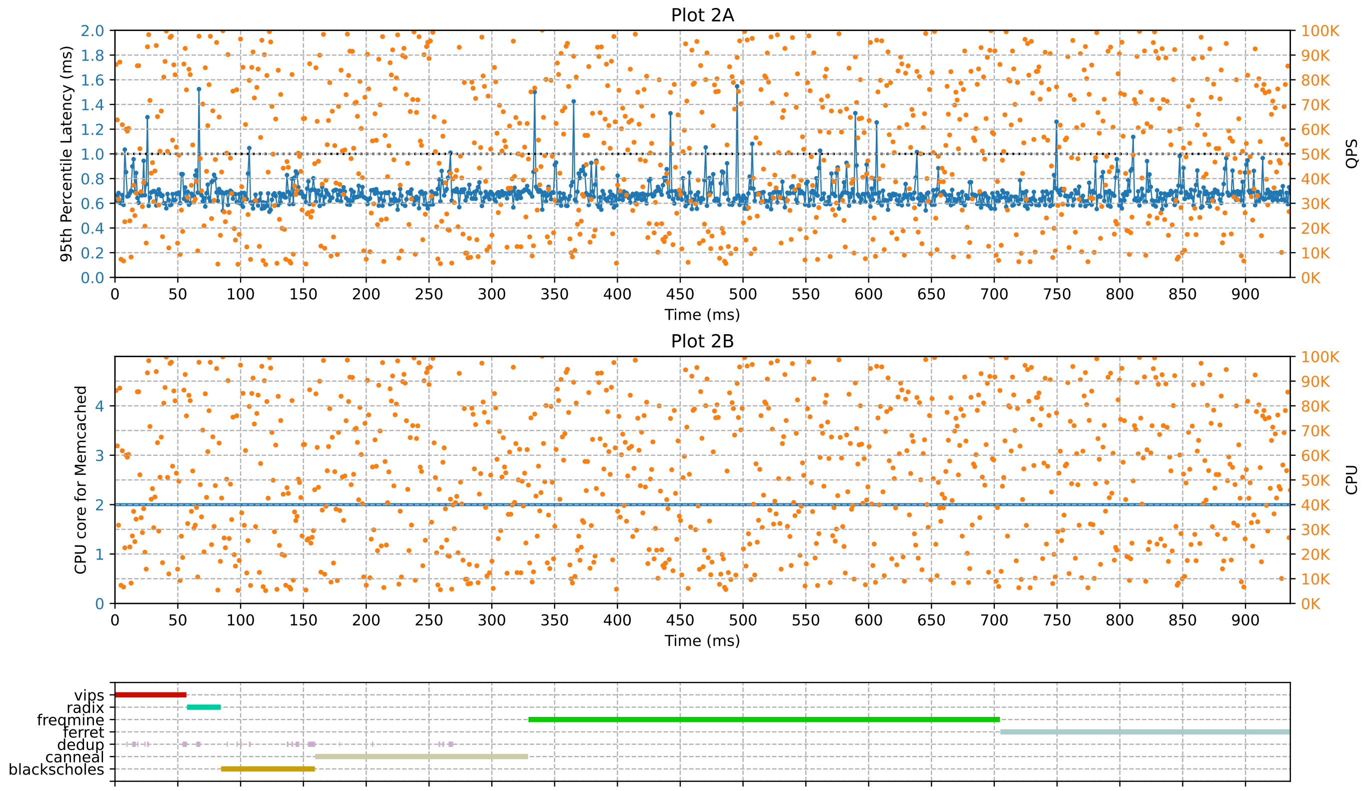


Figure 7: Run2: 95th percentile latency and CPU usage of *memcached* under dynamic load with frequency 1s. Together with the running time of jobs

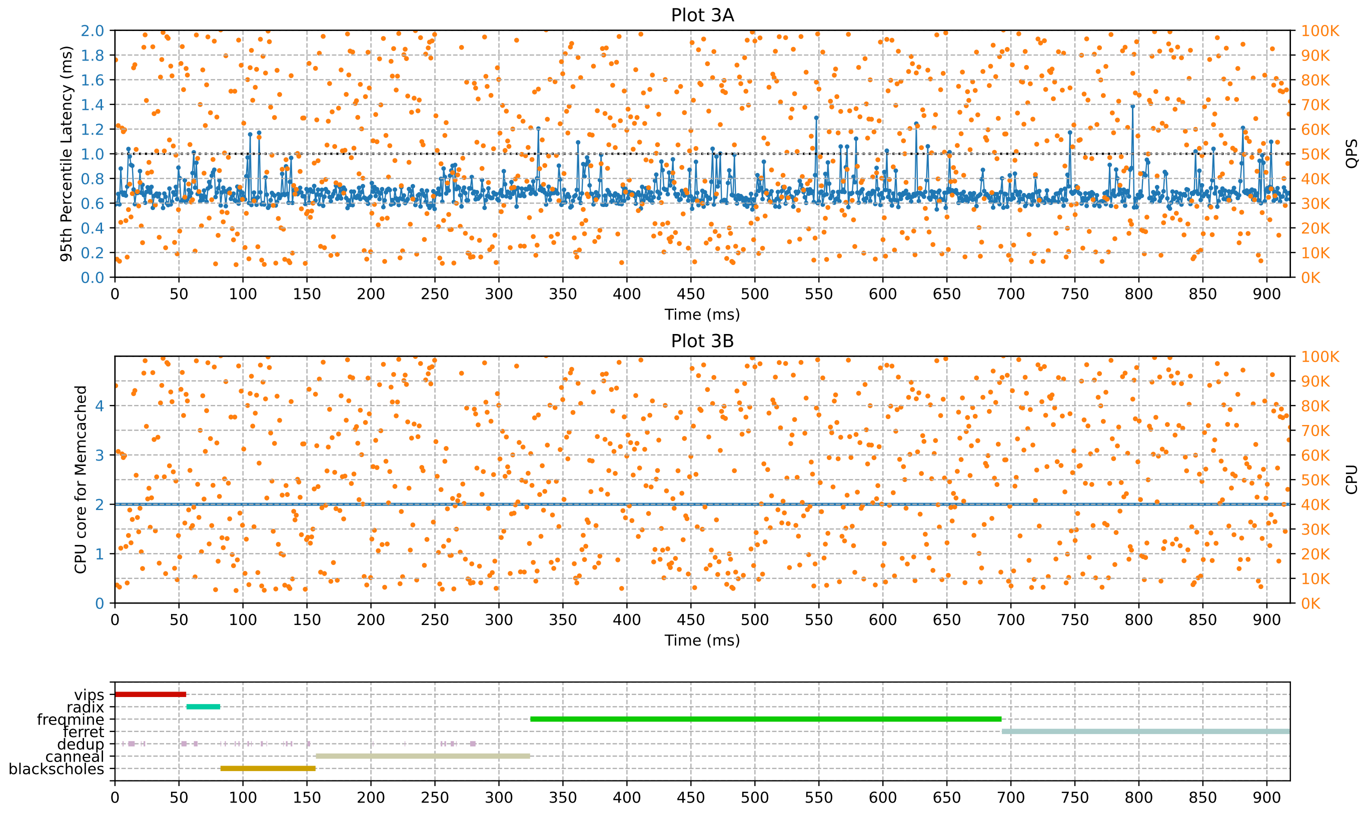


Figure 8: Run3: 95th percentile latency and CPU usage of *memcached* under dynamic load with frequency 1s. Together with the running time of jobs