Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Systems@ETH Zürich

Cloud Computing Architecture

Semester project report

Group 1
Andrea Keusch - 18-918-060
Shiduo Xin - 23-946-692
Yirui Zhang - 23-950-736

Systems Group
Department of Computer Science
ETH Zurich
May 17, 2024

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 batch job, as well as the latency outputs of memcached, running with a steady client load of 30K QPS. For each batch 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 - the makespan of 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. The SLO violation ratio should be calculated during the time from when the first batch-job-container starts running to when the last batch-job-container stops running.

job name	mean time [s]	std [s]
blackscholes	112.67	1.25
canneal	81.00	1.41
dedup	32.33	1.25
ferret	96.00	0.00
freqmine	115.33	16.03
radix	33.67	0.47
vips	37.00	0.82
total time	185.00	0.82

Answer:

Create 3 bar plots (one for each run) of memcached p95 latency (y-axis) over time (x-axis), with annotations showing when each batch job started and ended, also indicating the machine each of them is 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 in the bar plot, 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 and stopped, the blackscholes color to annotate when blackscholes started and stopped etc.

Plots:

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

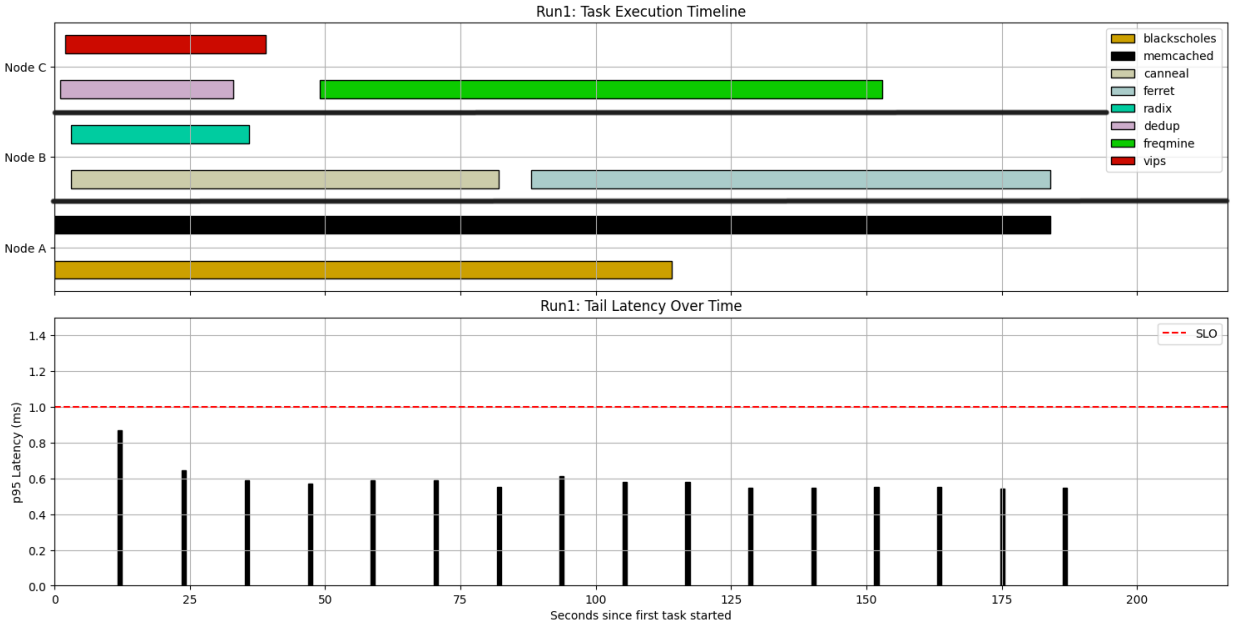

Figure 1: part3 run1

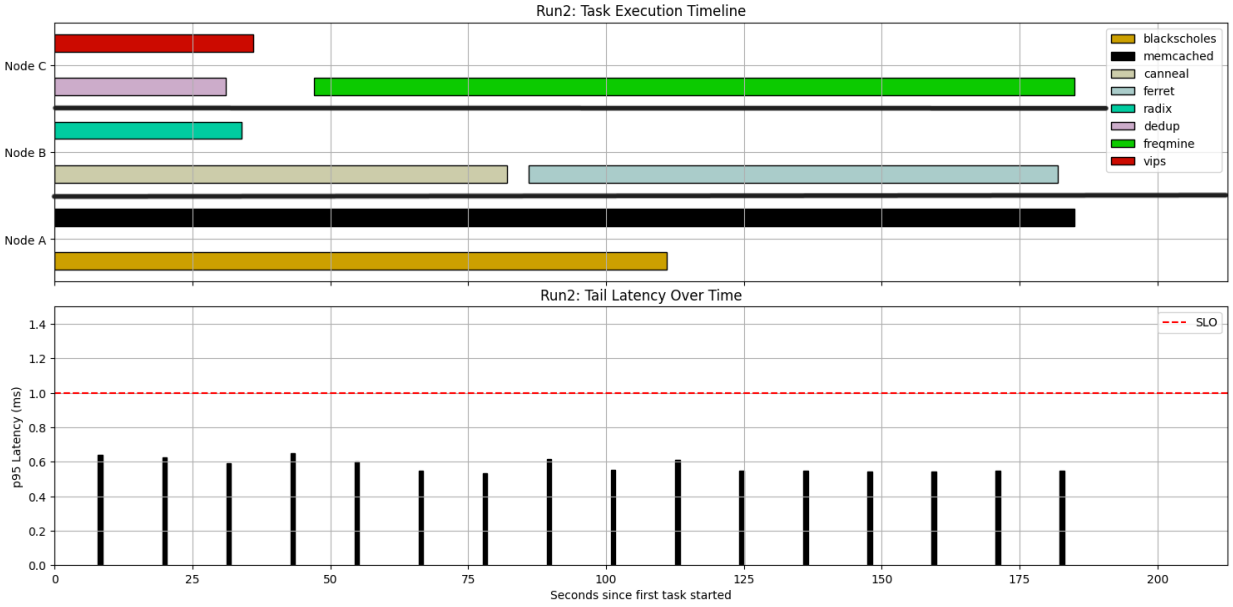

Figure 2: part3 run2

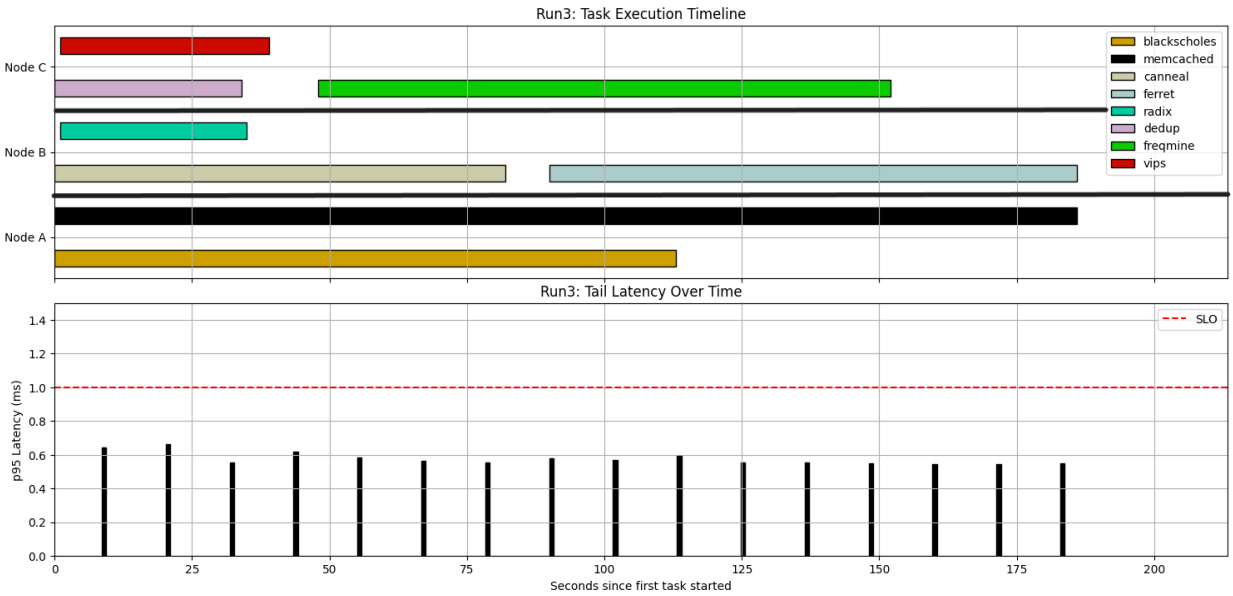

Figure 3: part3 run3

[SLO Violation Rate: 0]

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

   - Which node does memcached run on? Why?
     Answer: On node-a-2core, Memcached is sensitive to CPU source usage, it's not suitable to corroborate with most of the tasks, even with the CPU usage restriction, 5 in 7 tasks could make it exceed 1ms tail latency, seems it's because of llc which could not be controlled.

   - Which node does each of the 7 batch jobs run on? Why?
     Answer:
     - blackscholes: On node-a-2core, corroborates with Memcached. Analyzing the result from part12, blackscholes, canneal radix, which occupy little CPU l1i resources seem good to work with memcache. However, in experiments, radix would fail, and canneal run for more than 4 minutes or fail, and only blackscholes is good to corroborate with Memcached.
     - canneal: On node-b-4core, it starts at the beginning and corroborates with radix, from part2, we learn that canneal gains little from multi-cores and threads, but radix does, and these two can use up most of the resource and have close run time.
     - dedup: on node-c-8core, starts at the beginning and corroborates with vips, dedup speeds up little with multi threads, but vips does, so these two work together. seems dedup is a memory intensive work and best to fit in node-b, but I've tried to exchange its position with both canneal and radix, it turn out those run time are all longer.
     - ferret: On node-b-4core, ferret is heavy both in CPU and memory (and especially memory), and it speeds up well with multi-threads and cores, so it works alone after radix/canneal ends, and node-b could provide sufficient memory.

- freqmine: on node-c-8core, start after dedup/vips complete, freqmine speeds up a lot with multithread, so it works alone such that it can use the full resources.
- radix: On node-b-4core, just like that told in 'canneal', radix corroborating with canneal could uses up the left CPU resource.
- vips: on node-c-8core, corroborates with dedup and uses more CPU cores, they two can use up most resources and complete closely.

- Which jobs run concurrently / are colocated? Why?

  Answer:
  1. Blackscholes corroborates with Memcached. Analyzing the result from part1&2, blackscholes, canneal & radix, which occupy little CPU & l1i resources seem good to work with memcache. However, in experiments, radix would fail, and canneal run for more than 4 minutes or fail, and only blackscholes is good to corroborate with Memcached.
  2. dedup corroborates with vips, dedup speeds up little with multi threads, but vips does, so these two work together.
  3. canneal corroborates with radix, from part2, we learn that canneal gains little from multi-cores and threads, but radix does, and these two can use up most of the resource and have close run time.

- In which order did you run the 7 batch jobs? Why?

  Order (to be sorted):
  on node-a:blackscholes
  on node-b:(canneal+radix), ferret
  on node-c:(dedup+vips), freqmine

  Why: there are five jobs that start at the same time safely. ferret comes after canneal and radix finished, because it's heavy in memory and it may fail to collaborate with others. fremine comes after dedup and vips, because they would slow down each other when collaborating. I guess it could be TLB flushes frequently.

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

  Answer:

  - blackscholes: 2, it gains little with multithreads, and 2 threads is enough for it to end earlier than others. Leave cpu resource for a better tail latency.
  - canneal: 4, cannel gains well with 4 threads and it's a heavy work, in experiments it's safe to collaborate with radix with up to 4 threads.
  - dedup: 2, dedup speeds up well with 2 threads, but not if using more threads. Moreover, it could make it end up with vips at the same time, leaving all resource for freqmine.
  - ferret: 4, from part2, ferret speeds up well with 4 threads, and 8 threads increase only little.
  - freqmine: 8, with all cpu resource left, use as many threads as possible.
  - radix: 4, radix speeds up well with multi threads, but with 8 threads it slows down when run together with canneal, might be because of TLB/Cache flush.
  - vips: 4, a balance choice, to end up with dedup simultaneously.

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

  Answer: 1. adjusted all yaml file to assign cpu cores number and node type to jobs. 2. add a task3.sh script to run measurement automatically. 3. adjusted time.py to get all avg/std value in one go.

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

  Answer: 1. How to minimize the total time of nodes being idle. 2. how to let jobs speed up well with more threads get more threads and cores.

Please attach your modified/added YAML files, run scripts, experiment outputs and the report as a zip file. You can find more detailed instructions about the submission in the project description 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 [74 points]

1. **[18 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) [7 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. achieved 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.

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: See Figure 4. When memcached is bound to 1 core, using 2 threads instead of 1 has not effect on the saturation point but it increases the p95 latency. The explanation behind this is that using 2 threads causes overheads (such as context switches).
When memcached is allowed to use 2 cores, adding a second thread decreases the latency significantly and increases the saturation point. This is because we need 2 threads in order to utilize the resources of both cores.
Adding a second core has no effect if using only 1 thread since one thread cannot utilize 2 cores. However, if using 2 threads the benefits of having 2 cores are significant.

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 can be seen in a) only the combination of using 2 cores and 2 threads (T=2, C=2) satisfies the SLO latency at the highest load.

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: We need T=2 threads. With only 1 thread we would violate SLO no matter how many cores we use (see 1a). With 2 threads we can dynamically change between 1 or 2 cores depending on the current load, as discussed in 1a) and b).

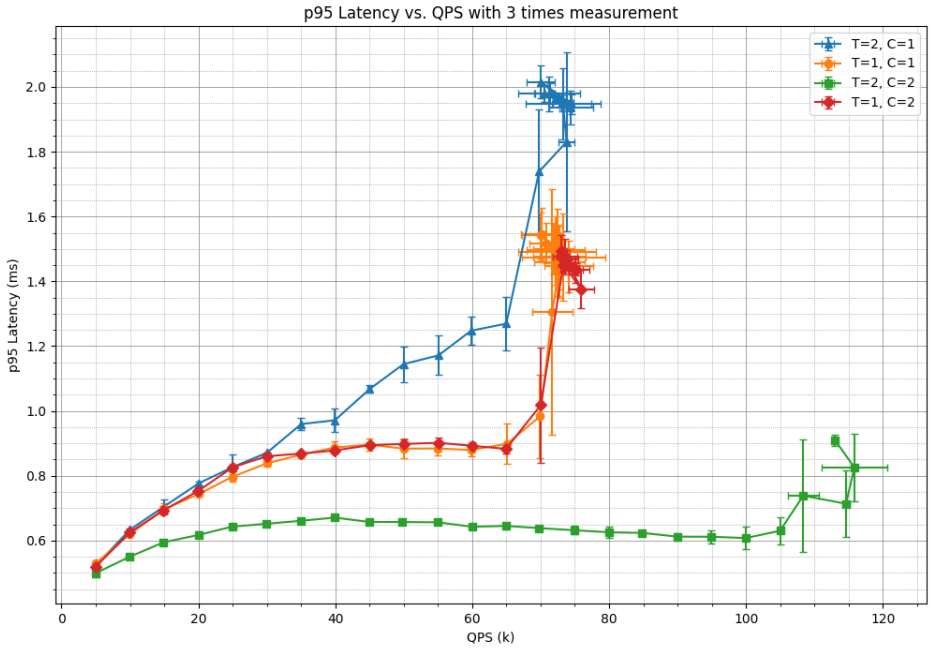

Figure 4: Plot for 4.1 a): The mean and standard deviation of the 95th latency measured over 3 runs.

d) [8 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 achieved 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. We averaged over 3 runs. For the CPU utilization, we measured the utilization every second. For each target QPS measurement, we extracted all corresponding CPU measurements (using timestamps to correlate the data), discarded the first one since that might be inaccurate and then averaged over the other 4 measurements. This gives us the CPU utilization of one run. We averaged this and the p95 latency over 3 runs.

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 \
```

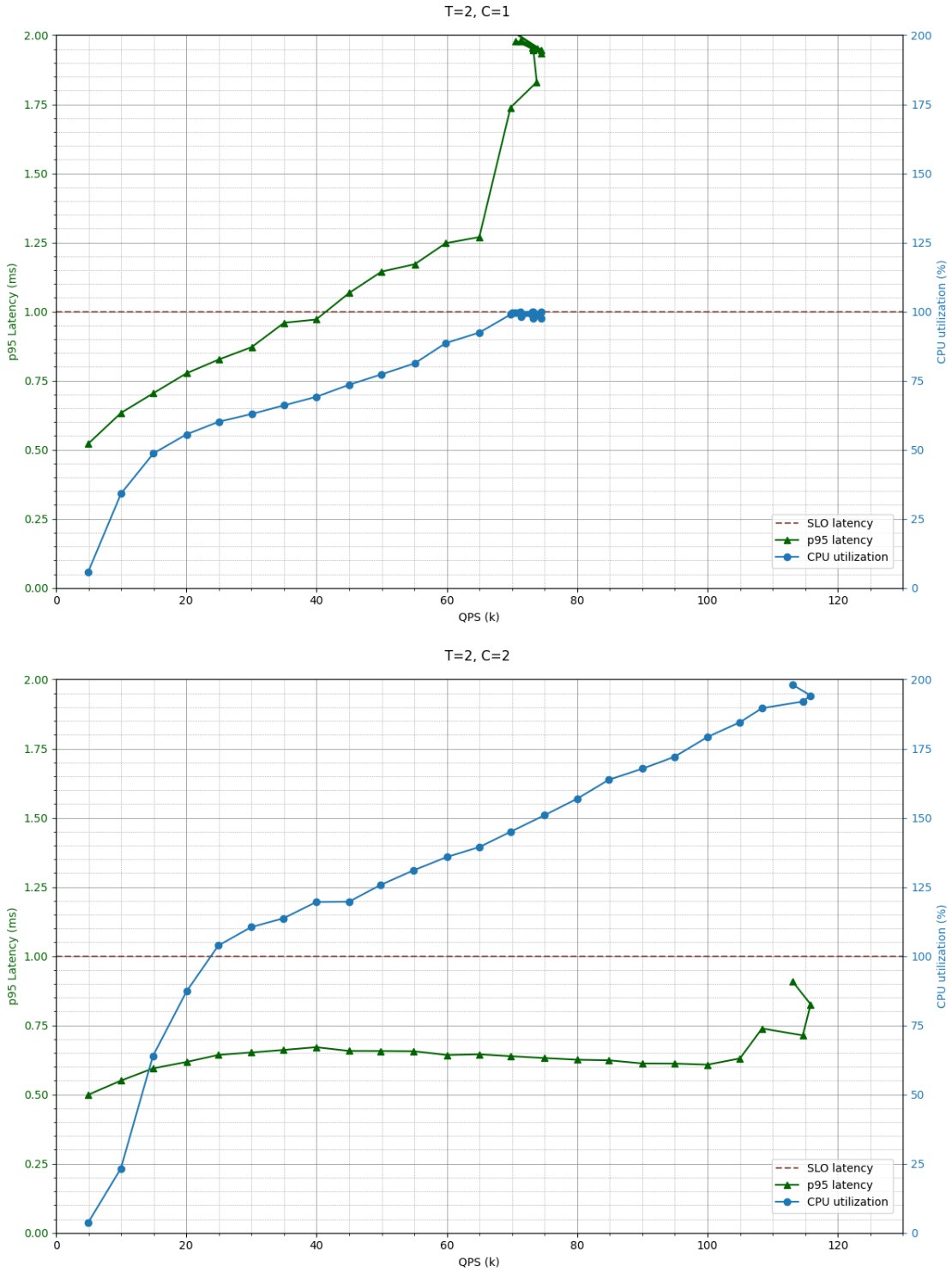

Figure 5: Plots for 4.1d: The p95 latency and CPU utilization averaged over 3 runs.

```
--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 benchmarks (batch jobs) on the 4-core VM. The goal of your scheduling policy is to successfully complete all batch 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 batch 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 batch jobs complete successfully and do not crash. Note that batch 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: Memcached is running on core 0 or cores 0 and 1, depending on the load. The load is determined by monitoring the CPU utilization of the memcached cores. We have two queues of batch jobs. The "large" jobs can only run when memcached load is low. They run on cores 1-3. The "small" jobs run on cores 1-3 or 2-3 depending on memcached load. In the beginning, we run a large job whenever the load on memcached is low and a small job otherwise. Once all large jobs finished, we always keep a small job running. At all times, there is at most 1 job running and at most one job paused. The distinction between large and small job was made based on the llc and memory requirements of the job since those resources are shared with memcached cores.

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

  Answer: When CPU utilization of the memcached cores is low, we assign 1 core. If it is higher, we assign 2 cores. We chose this metric, because the CPU utilization is a good measurement on how big the current load on memcached is (see part 4.1). We experimented with different thresholds and decided on one that reduces SLO violations.

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

  Answer:

  - blackscholes: 2 cores as it is not that resource intensive and we treat it as a small job
  - canneal: 2 cores as it is not that resource intensive and we treat it as a small job
  - dedup: 3 cores, because its resource intensive (especially on llc, memBW and l1i)
  - ferret: 3 cores, because its long running and resource intensive (especially on llc and memBW)

  - freqmine: 2 or 3 cores, because its resource intensive and long-running but can still run fine on 2 cores
  - radix: 2 cores as it is not that resource intensive and we treat it as a small job
  - vips: 2 cores as it is not that resource intensive and we treat it as a small job

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

  Answer:

  - blackscholes: 2 cores because it always runs on only 2 cores anyways hence using more than 2 threads would just add overhead. 2 threads is better than 1 thread (seen in part 2.2)
  - canneal: same as blackscholes
  - dedup: 4 threads since it runs on 3 cores hence it can benefit from having more than 2 threads. Using 4 threads is better than 2 (seen in part 2.2)
  - ferret: same as dedup
  - freqmine: 4 threads since it might run on 3 cores and then benefits from more than 2 threads (sames as dedup)
  - radix: same as blackscholes
  - vips: same as blackscholes

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

  Answer: We do not collocate any jobs since we found that the CPU is mostly fully utilized for the job batches with our core assignments and policy. Hence, we do not expect to see any improvement when collocating jobs.

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

  Order (to be sorted):

  Small jobs: blackscholes, radix, vips, canneal, freqmine

  Large jobs: ferret, dedup

  Why:

  Small jobs are more or less order by increase memory requirements (llc and mem). Large jobs are ordered by decreasing memory requirements. Freqmine is the last small job since, once all large jobs finished, it might be able to use 3 cores. As it is the longest running and most intensive job, we think running this on 3 cores has the most benefit (out of all small jobs).

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

  Answer: In part 3, we had a 8-core node where we were able to collocate jobs. We do not have such a node in part 4. Moreover, part 3 had a 4-core node fully available to the jobs, which we also do not have. Hence, we do not collocate any jobs. Maybe we could have collocated blackscholes with memcached (or one of the jobs) in some situations. But since the QPS can be much higher as in part 3, we did not explore this option. Maybe we could have collocated 2 jobs in some situations. But we did not explore this option due to the credit limit.

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

  Answer: We implemented the scheduler as Python script. For the batch jobs, we used Docker Python SDK and its provided container functionality such as run, pause, unpause, update(cpuset_cpus=...) and remove. For memcached, we use the command `sudo taskset -a -cp [cores] [pid]` command, executed via subprocess.run(). We require that memcached and mcperf are already running.

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

  Answer: We considered that having a high load on llc and mem could put inference on memcached because those resources are shared between cores. We also considered that memcached highly uses the CPU und l1i cache which is one reason why we did not collocate any jobs with memcached (since those jobs would add inference on those resources). Also, we tried a strategy to avoid switching back and forth too often. Whenever memcached runs on 2 cores (high load), it takes 5 iterations of our main loop until it can switch back to 1 core (low load).

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 batch job 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 batch job, as well as the latency outputs of memcached. For each batch application, compute the mean and standard deviation of the execution time [2] across three runs. Compute the mean and standard deviation of the total time to complete all jobs - the makespan of 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. The SLO violation ratio should be calculated during the time from when the first batch-job-container starts running to when the last batch-job-container stops running.

Answer: SLO violations: run 1: 2.8%, run 2 = 5.5%, run 3 = 7.0%

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 benchmark (batch) application starts executing at which time. If you pause/unpause any workloads as part of your policy,

[2] Here, you should only consider the runtime, excluding time spans during which the container is paused.

job name	mean time [s]	std [s]
blackscholes	55.9	2.80
canneal	164.5	3.87
dedup	12.88	8.79
ferret	121.46	2.13
freqmine	125.39	48.11
radix	14.14	9.65
vips	39.36	10.96
total time	716.10	3.20

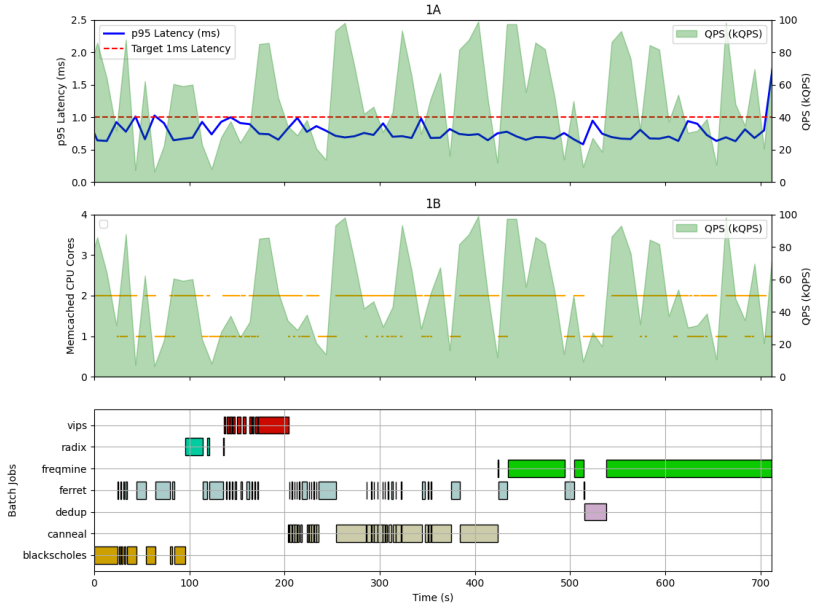

Figure 6: Task 4.3 - run 1 (Plots 1A, 1B)

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 Figures 6, 7 and 8

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
```

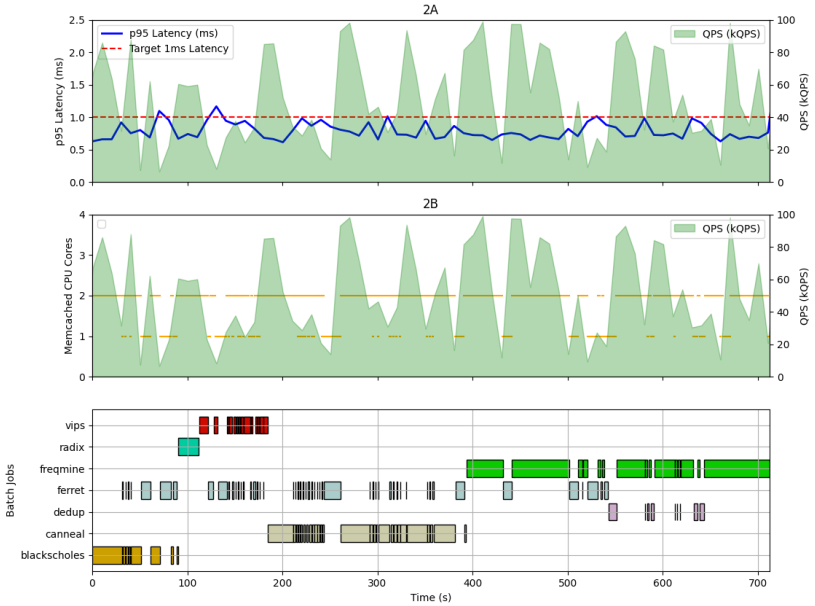

Figure 7: Task 4.3 - run 2 (Plots 2A, 2B)

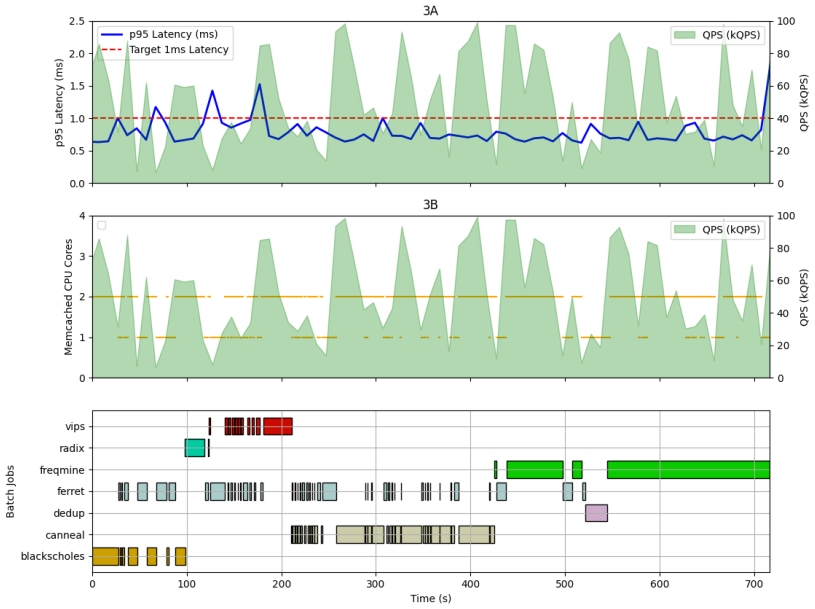

Figure 8: Task 4.3 - run 3 (Plots 3A, 3B)

```
$ ./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: The total runtime is comparable to the one from 4.3. However, the ratio of SLO violations seems to be a bit higher and sometimes it switches between 1 and 2 cores for memcached more often, which is what we expect.

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? The SLO violation ratio should be calculated during the time from when the first batch-job-container starts running to when the last batch-job-container stops running.

Answer: run 1: 5.7%, run 2: 9.7%, run 3: 9.9%

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: Theoretically, an interval $> 6.6s$ will do. But in part 4.3 we are already above the 3% SLO violation ratio.

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

Answer: In order to measure the CPU utilization properly, we sleep 200ms between two iterations of our loop (e.g. between making decisions). The interval has to be larger than that for sure. Since we might violate the SLO in those 200ms, the interval needs to be even higher than that. Assume we always violate the SLO during the first 200ms after the load changed, than we would need an interval of $>$ 6.6s $(200/3 * 100)$.

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.

job name	mean time [s]	std [s]
blackscholes	34.63	4.64
canneal		
dedup	12.88	8.78
ferret	48.66	1.33
freqmine	184.69	36.83
radix		
vips		
total time	971.81	0.47

Plots:

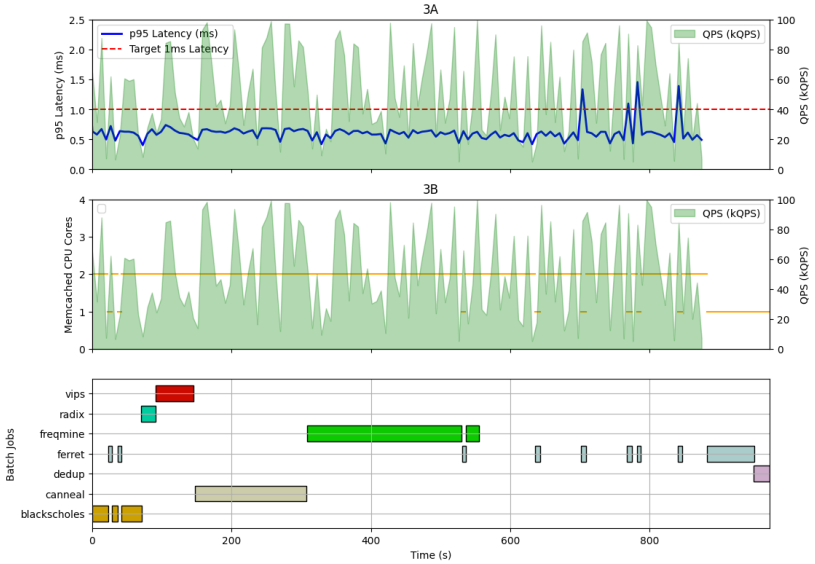

Figure 9: Task 4.4 - run 1 (Plots 1A, 1B)

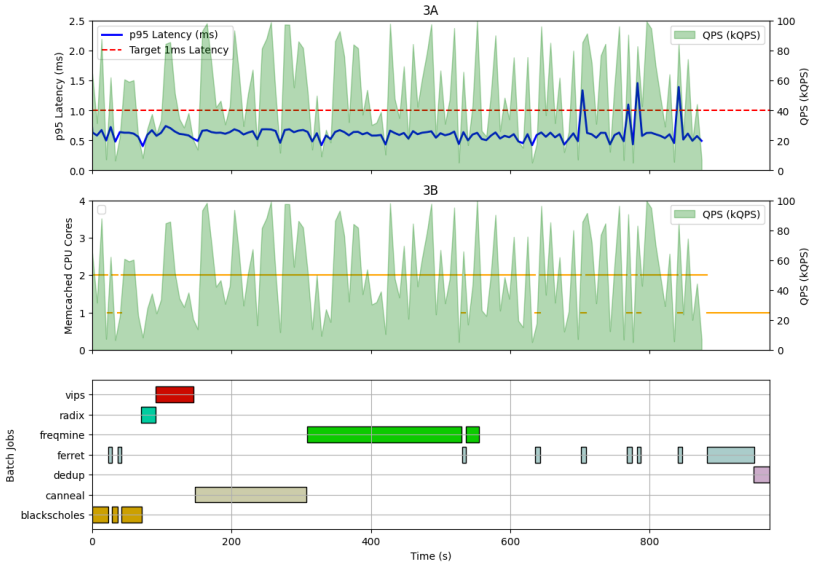

Figure 10: Task 4.4 - run 2 (Plots 2A, 2B)