Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Systems@ETH Zürich

Cloud computing architecture

Semester project report

Group 027
Jiale Chen - 20-961-504
Ran Liao - 20-949-186
Xintian Yuan - 20-951-778

Systems Group
Department of Computer Science
ETH Zurich
May 26, 2021

Instructions

Please do not modify the template, except for putting your solutions, names and legi-NR.

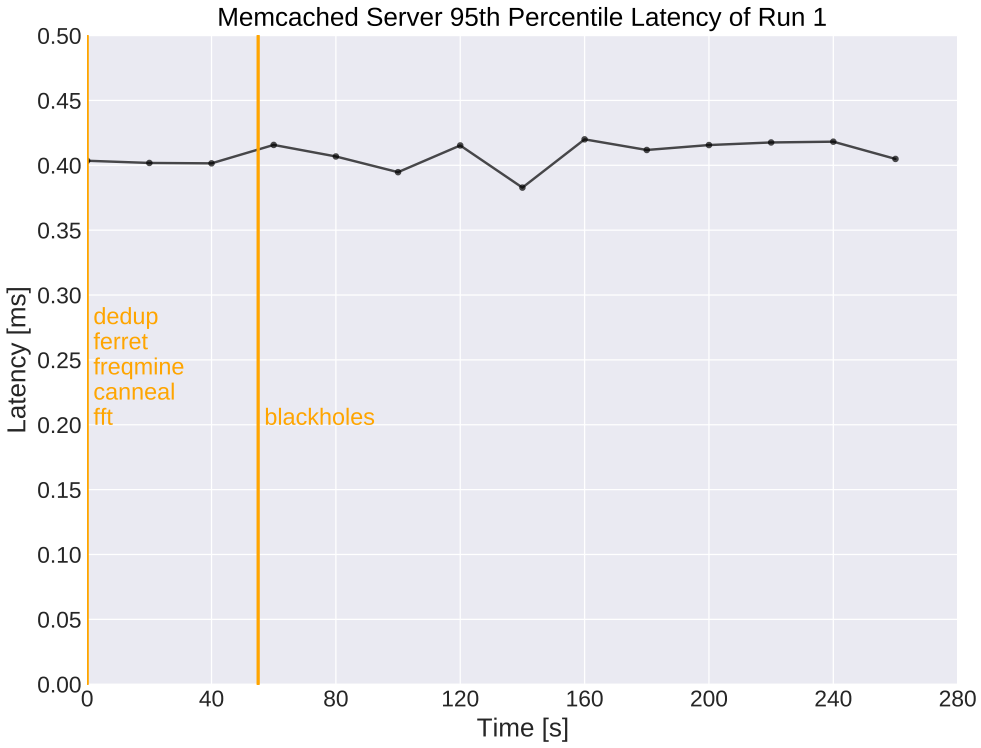
Part 3 [34 points]

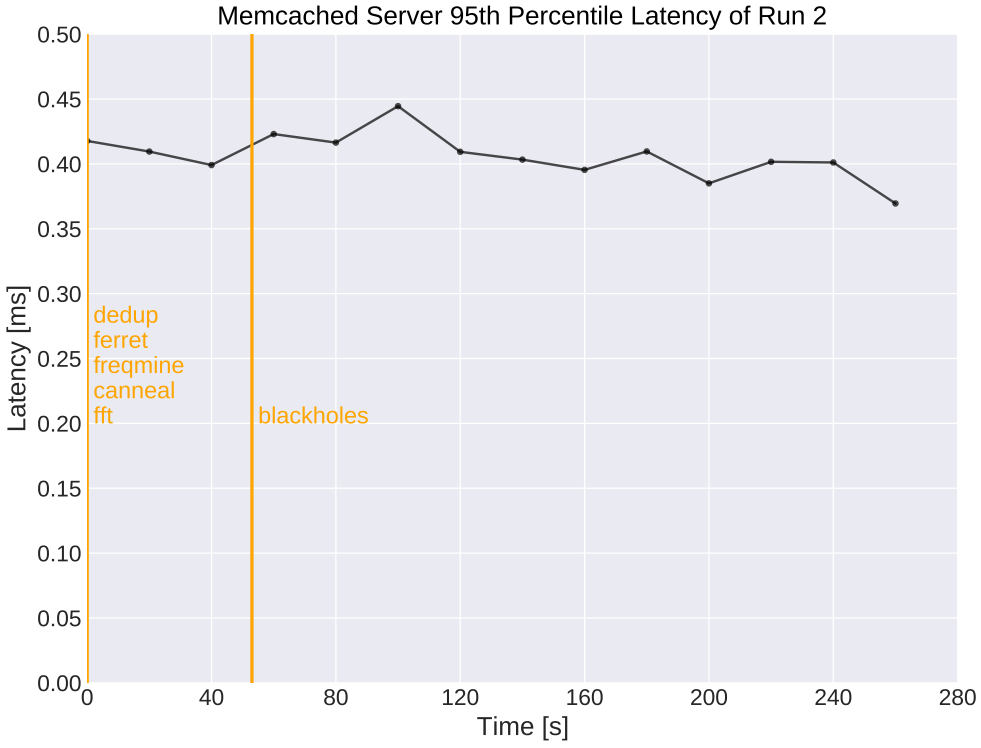
1. **[17 points]** With your scheduling policy, run the entire workflow 3 separate times. For each run, measure the execution time of each PARSEC job, as well as the latency outputs of memcached running with a steady client load of 30K QPS. For each PARSEC application, compute the mean and standard deviation of the execution time 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 datapoints with 95th percentile latency $>$ 2ms, as a fraction of the total number of datapoints. Do three plots (one for each run) of memcached p95 latency (y-axis) over time (x-axis) with annotations showing when each parsec job started.

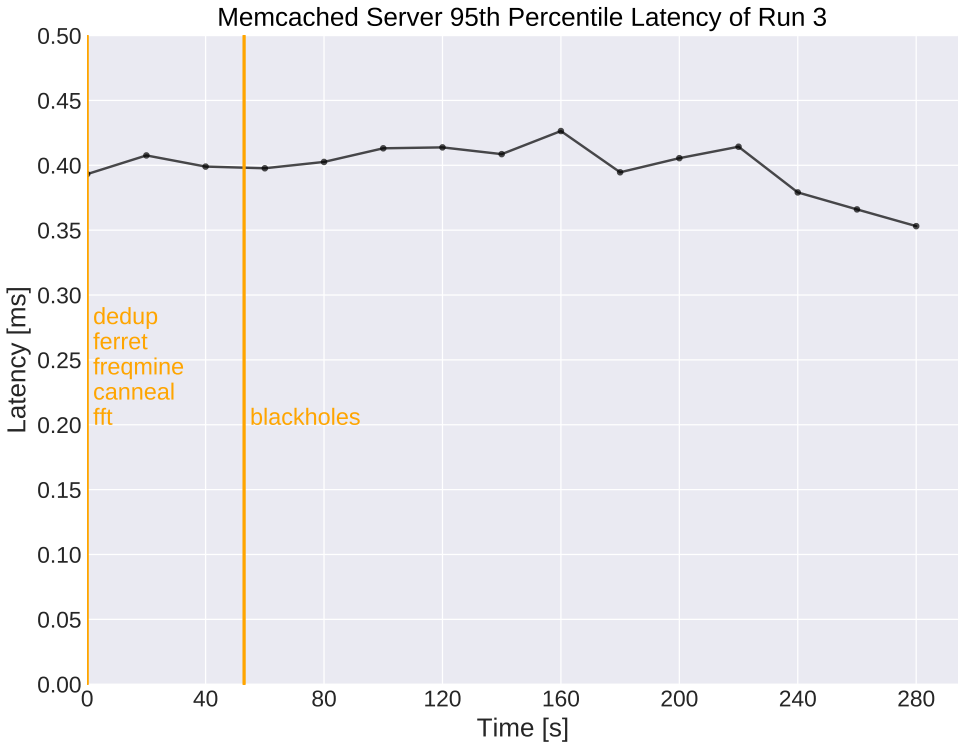
   Solution:

   The SLO violation ratio for all three runs are 0. The orange line annotates when each PARSEC job starts. The plots are only drawn to the point where all PARSEC jobs finished.

job name	mean time [s]	std [s]
dedup	53.67	0.94
blackscholes	195.67	1.25
ferret	261	6.38
freqmine	267	8.52
canneal	278	12.75
fft	144.67	12.97
total time	280.33	13.22


Memcached Server 95th Percentile Latency of Run 1
Latency [ms]
0.00
0.05
0.10
0.15
0.20
0.25
0.30
0.35
0.40
0.45
0.50
dedup
ferret
freqmine
canneal
fft
blackholes
0
40
80
120
160
200
240
280
Time [s]


Memcached Server 95th Percentile Latency of Run 2
Latency [ms]
0.00
0.05
0.10
0.15
0.20
0.25
0.30
0.35
0.40
0.45
0.50
dedup
ferret
freqmine
canneal
fft
blackholes
0
40
80
120
160
200
240
280
Time [s]



2. **[17 points]** Describe and justify the "optimal" scheduling policy you have designed. This is an open question, but you should at minimum answer the following questions:

   - Which node does memcached run on?
   - Which node does each of the 6 PARSEC apps run on?
   - Which jobs run concurrently / are collocated?
   - In which order did you run 6 PARSEC apps?
   - How many threads you used for each of the 6 PARSEC apps?

   Describe how you implemented your scheduling policy. Which files did you modify or add and in what way? Which Kubernetes features did you use? Please attach your modified/added YAML files, run scripts and report as a zip file. **Important: The search space of all the possible policies is exponential and you do not have enough credit 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.**

   Solution:

   Our solution distributed the seven jobs (one memcached and 6 PARSEC) on different nodes with various cores. We use nodeSelector to run jobs only on specified node. We use taskset to restrict each job only running on specified cpu cores. See the following table for details.

job name	node	cpu id	number of threads
dedup	2-cores-node	0-1	2
blackscholes	2-cores-node	0-1	2
freqmine	4-cores-node	0-3	4
fft	4-cores-node	0-3	4
memcached	8-cores-node	0	1
ferret	8-cores-node	1-7	7
canneal	8-cores-node	4-7	4

At first, dedup, ferret, freqmine, canneal and fft starts simultaneously and runs concurrently, then blackscholes will start after dedup is completed. This means memcached, canneal and ferret are collocated on 8-cores-node. At the same time fft and freqmine are collacted on core 4-cores-node. But dedup and blackscholes will not collcated with each other and will run sequentially. We used blackscholes with 2 threads, canneal with 4 threads, dedup with 2 threads, ferret with 7 threads, fft with 4 threads and freqmine with 4 threads(1 thread for each cpu core they can use). We modified the respective YAML file to the configuration mentioned above and add a scheduler.py python script to automate the scheduling process.

We put memcached into the 8-cores-node because from part 2.3 we learned that all PARSEC jobs are sub-linear with more than 4 threads. This means give a job with 7 threads or 8 threads doesn't make much difference. In 8-cores-node machine, memcached will use 1 core, but we still can execute jobs on the remaining 7 cores. ferret is quite time consuming among all PERSEC jobs and become the bottleneck somehow. So we run it with 7 threads, this is almost the maximum possible resource we can allocate to this job.

And to maximize resource utilization, we try to let jobs in 2-cores-node, 4-cores-node and 8-cores-node complete approximately the same time.

Part 4 [76 points]

1. **[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

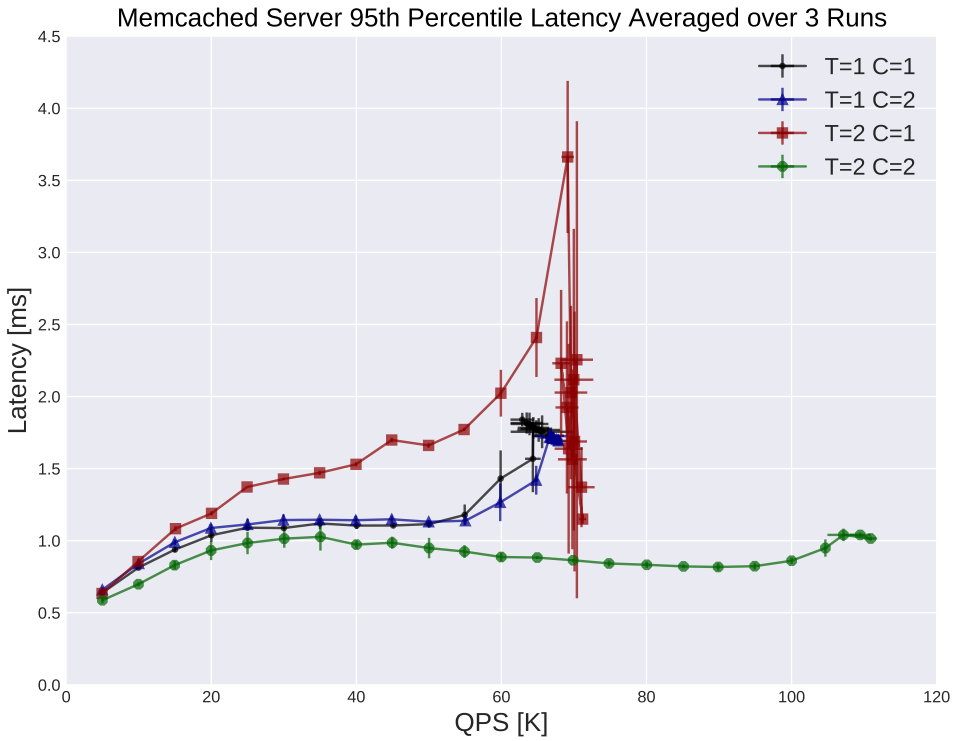
   For this question, use the following `mcperf` command to vary QPS from 5K to 120K:

```
$ ./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 5000:120000:5000
```

   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.

   Solution:



Performance with 2 threads and 2 cores (green line) is significantly better than other combinations. One core is not enough for memcached to support QPS larger than 60k-70k. If we use only 1 thread, giving it more cores doesn't help significantly. And if we use 2 thread with only 1 core, additional context switching between these 2 threads will damage the performance even further.

2. **[8 points]** Now assume you need to support a memcached request load that ranges from 5K to 100K QPS while guaranteeing a 2ms 95th percentile latency SLO.

   a) To support the highest load in the trace (100K QPS) without violating the 2ms latency SLO, how many memcached threads (T) and CPU cores (C) will you need? i.e., what value of T and C would you select?

   Solution:

   I will use 2 threads and 2 cores. Other combinations do not support 100K QPS.

   b) Assume you can change the number of cores allocated to memcached dynamically as the QPS varies from 5K to 100K, 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 2ms 95th percentile latency SLO while the load varies between 5K to 100K QPS? i.e., what values of T would you select?

   Solution:

   I will use 2 threads. We can support 100K QPS only when 2 threads are used.

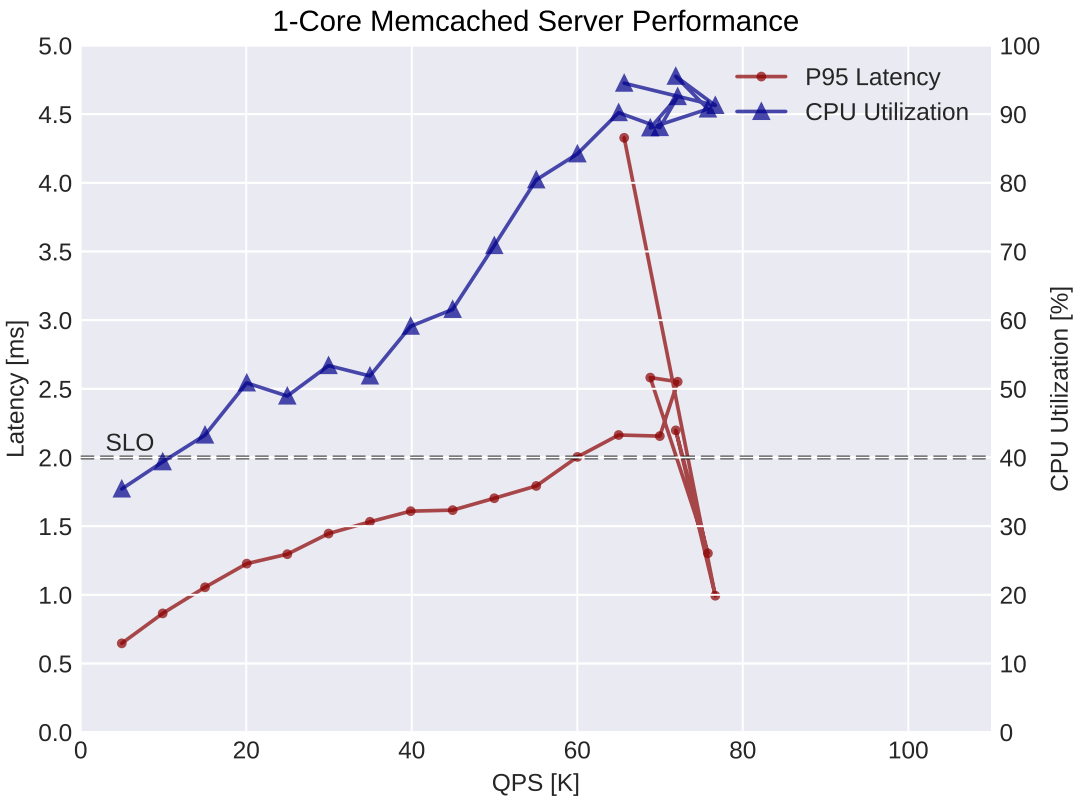
   c) Run memcached with the number of threads T that you proposed in b) above and measure performance with $C = 1$ and $C = 2$. Use the following `mcperf` command to sweep QPS from 5K to 100K:

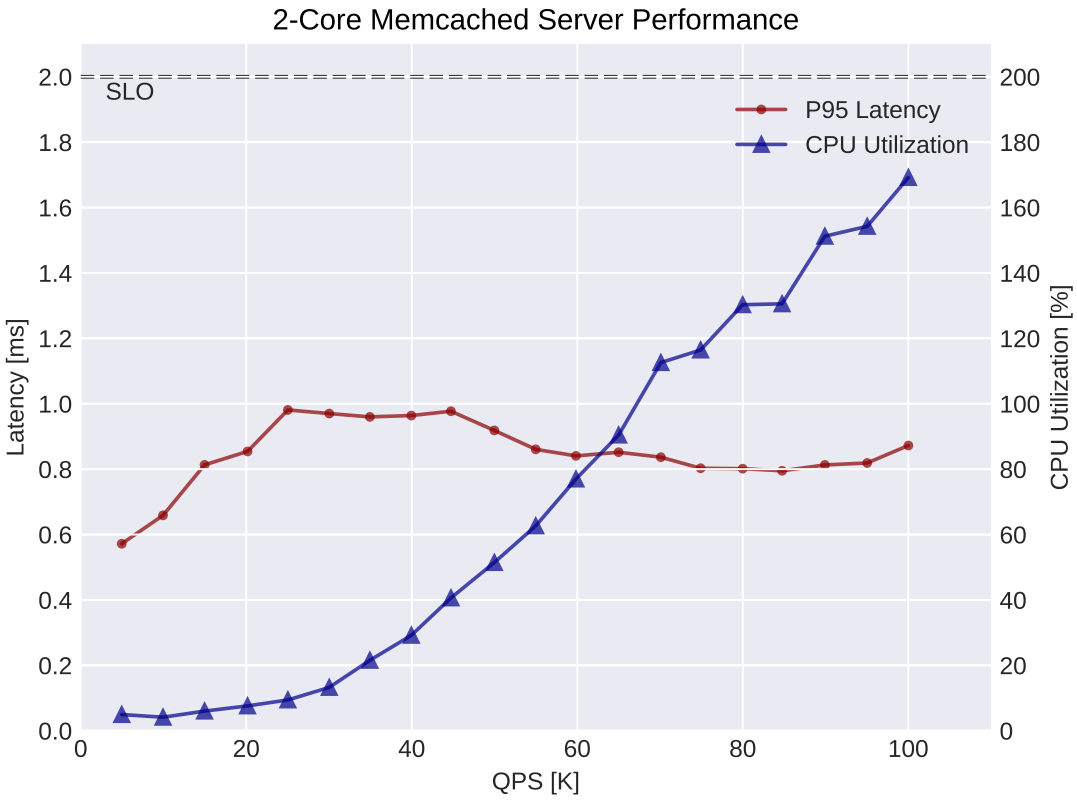
```
$ ./mcperf -s INTERNAL_MEMCACHED_IP --loadonly
$ ./mcperf -s INTERNAL_MEMCACHED_IP -a INTERNAL_AGENT_IP  \
           --noload -T 16 -C 4 -D 4 -Q 1000 -c 4 -t 5 \
           --scan 5000:100000:5000
```

   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 5K to 100K. In each graph, use two y-axes. Plot the 95th percentile latency on the left y-axis. Draw a dotted horizontal line at the 2ms 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.

   Solution:


1-Core Memcached Server Performance
P95 Latency
CPU Utilization
SLO
Latency [ms]
CPU Utilization [%]
QPS [K]
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
4.5
5.0
0
10
20
30
40
50
60
70
80
90
100
0
20
40
60
80
100


2-Core Memcached Server Performance
SLO
P95 Latency
CPU Utilization
Latency [ms]
CPU Utilization [%]
QPS [K]
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6
1.8
2.0
0
20
40
60
80
100
120
140
160
180
200
0
20
40
60
80
100

3. [**15 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.

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 2ms 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.** 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. To describe your scheduling policy, you should at minimum answer the following questions. For each, also **explain why**:

- How do you decide how many cores to dynamically assign to memcached?
- How do you decide how many cores to assign each PARSEC job?
- How many threads do you use for each of the PARSEC apps?
- Which jobs run concurrently / are collocated and on which cores?
- In which order did you run the PARSEC apps?
- How does your policy differ from the policy in Part 3?
- How did you implement your policy? e.g., docker cpu-set updates, taskset updates for memcached, pausing/unpausing containers, etc.

Solution:

Based on question 2(b), we decide to run memcached using 2 threads. If CPU usage of memcached is higher than 90%, we assign 2 cores to it. If the usage drops below 90%, we assign only 1 core to it. We choose 90% as the threshold because, according to the graphs of question 2(c), it needs to change to 2 cores when the QPS is greater than 60K, and the corresponding CPU usage for QPS greater than 60K is above 90%.

The VM has 4 cores in total. When memcached uses 2 cores, we use the remaining 2 cores to run PARSEC jobs. When memcached uses only 1 core, we use the remaining 3 cores to run PARSEC jobs. We have observed that the interference is very high when 2 PARSEC jobs are running at the same time. It costs more time to run in parallel than to run sequentially. So, our basic idea is to run the jobs sequentially with 3 threads for each job. However, fft does not support 3 threads. We also find that some jobs are not fully parallelizable, which has a long preprocessing or postprocessing stage when they can only run with 1 thread. As an optimization, we run fft in 2 threads and use pause/unpause function to let fft run only when other jobs are in single thread stage. In this way, we can fully utilize the CPU cores by

ensuring there are 3 threads running nearly all the times. The 3 threads are always running in the same CPU set (2 or 3 cores). We do not assign a dedicated core to a specific job, so that we have more flexibility to deal with different random seeds.

The picture below depicts our schedule policy. For simplicity, the time is not drawn proportional to real time. Core 0 and 1 are assigned to PARSEC jobs, core 3 is assigned to memcached, and core 2 is dynamically assigned. We run dedup, blackscholes, canneal, ferret, and freqmine sequentially. And we also let fft run in the single thread stages of dedup, blackscholes, and canneal. We choose this order because the first 3 jobs have a longer single thread stage which can allow fft to finish, while on the other hand, ferret and freqmine almost do not have a single thread stage.

The main difference to part 3 is the core allocation is dynamically adjusted to adapt the changing QPS of memcached. Moreover, because we only use one VM in part 4, the jobs are mainly running in sequence instead of in parallel.

We write our scheduler in python. We check the status and perform scheduling operations every 0.5 second, which is small enough compared to the 10 seconds interval. We use Docker Python SDK for PARSEC jobs (start/remove jobs, update CPU sets, pause/unpause jobs, collect logs, etc.). We execute taskset command to change CPU sets for memcached. We execute top command to get precise CPU usage of memcached and pid of PARSEC jobs. We read /proc/<pid>/status files to get the current number of threads of PARSEC jobs.

4. **[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 42
```

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. Also 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 datapoints with 95th percentile latency $>$ 2ms, as a fraction of the total number of datapoints.

Solution:

SLO violation ratio:
Run 1: 0 / 138 = 0 %
Run 2: 0 / 137 = 0 %

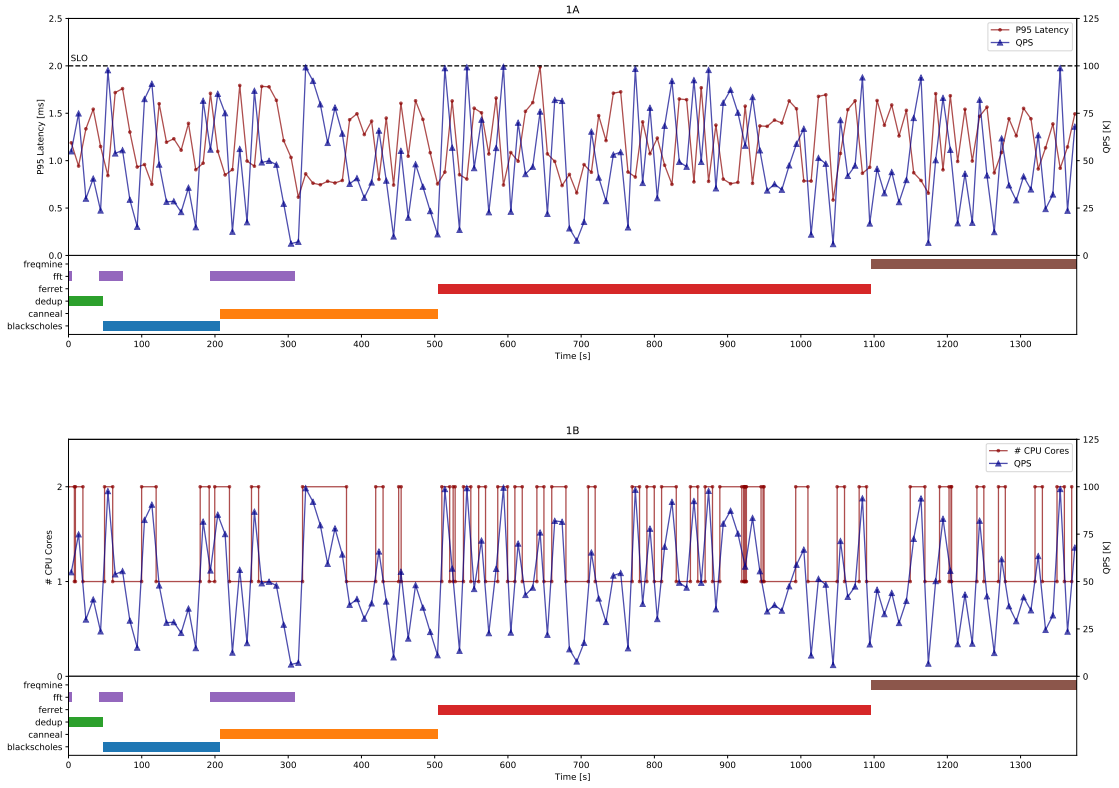
job name	mean time [s]	std [s]
dedup	45.8	1.16
blackscholes	155.7	3.11
ferret	586.5	3.13
freqmine	278.1	2.07
canneal	295.2	1.74
fft	149.9	3.78
total time	1363.9	11.04

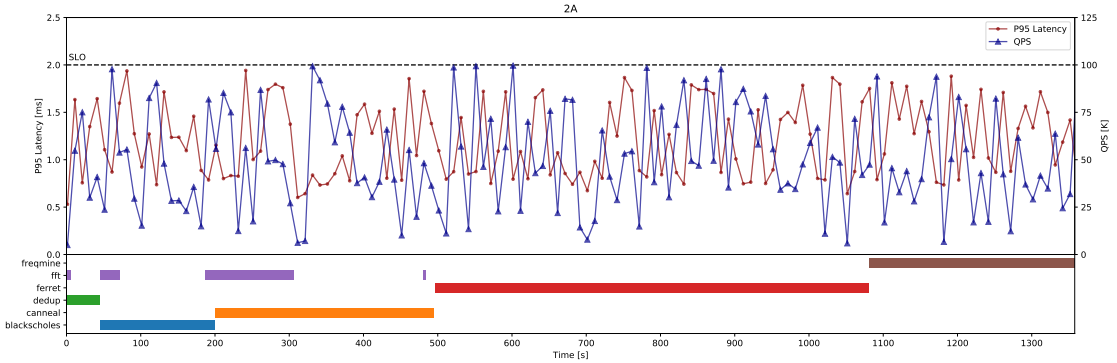
Run 3: 2 / 136 = 1.5 %

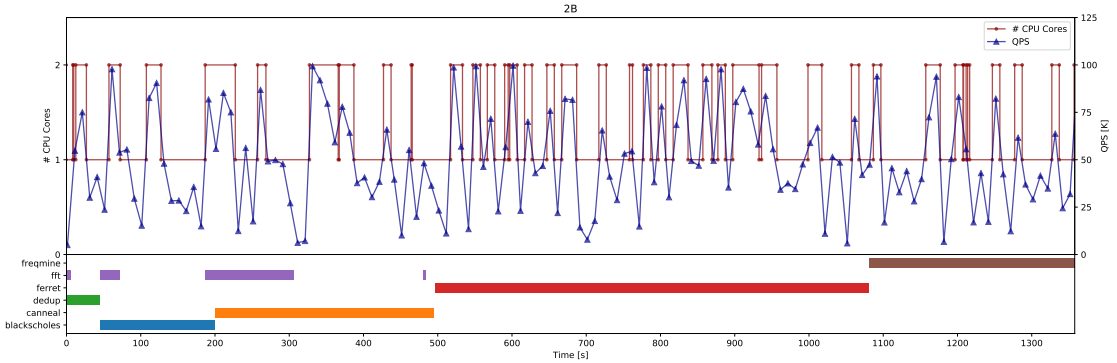
Include six plots – two plots for each of the three runs – with the following information. Label the plots as 1A, 1B, 2A, 2B, 3A, and 3B where the number indicates the run and the letter indicates the type of plot (A or B), which we describe below. In all plots, time will be on the x-axis and you should 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.

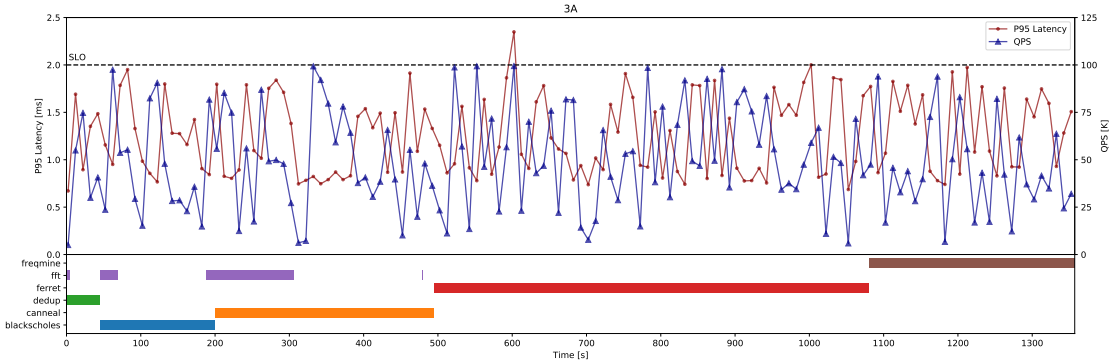
Solution:

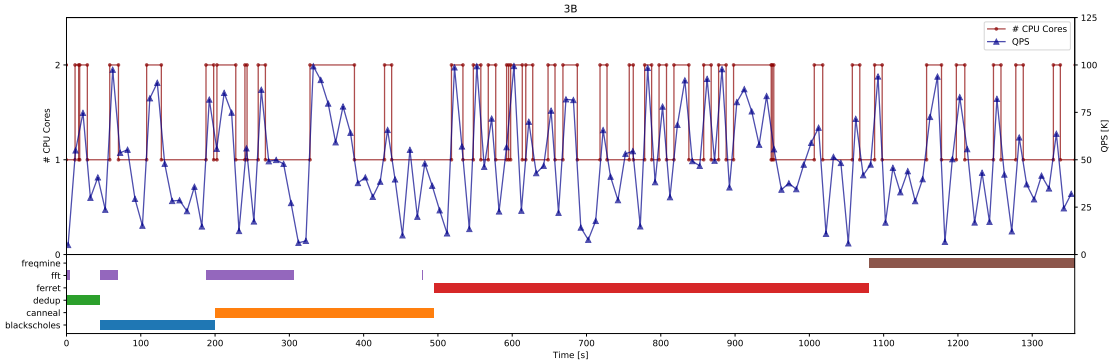
The horizontal bar below the graph indicates the time range when a PARSEC job is running.











5. **[20 points]** Repeat Part 4 Question 4 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 42
```

You do not need to include the plots or table from Question 4 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 4. What is the SLO violation ratio for memcached (i.e., the number of datapoints with 95th percentile latency $>$ 2ms, as a fraction of the total number of datapoints) with the 5-second time interval trace?

Solution:

The average total running time is 1396.8 seconds, slightly larger than that of the 10-second interval. The CPU core allocation is changed more frequently to adapt the faster changing QPS load, which may cause some overhead.

The SLO violation ratio becomes larger. With more variations in QPS, there are more chances that the scheduler may not be able to react immediately when the QPS suddenly increase, and so SLO is violated more often.

SLO violation ratio:
Run 1: 1 / 277 = 0.4 %
Run 2: 5 / 281 = 1.8 %
Run 3: 1 / 283 = 0.4 %

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%? 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 4.

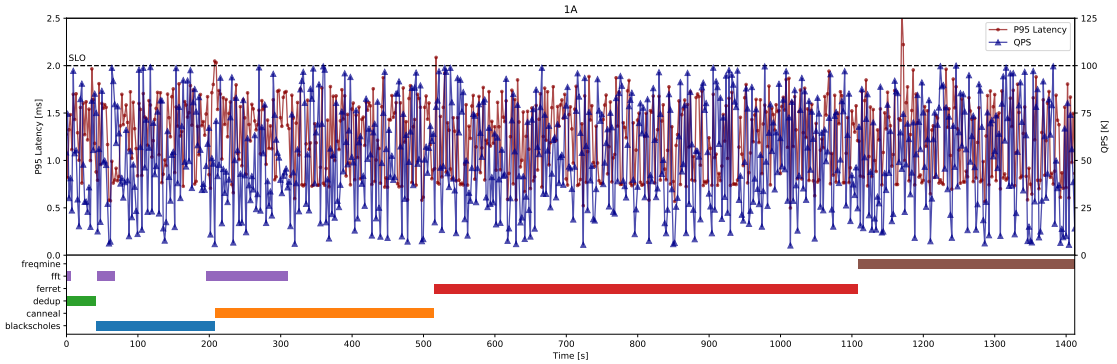
Solution:

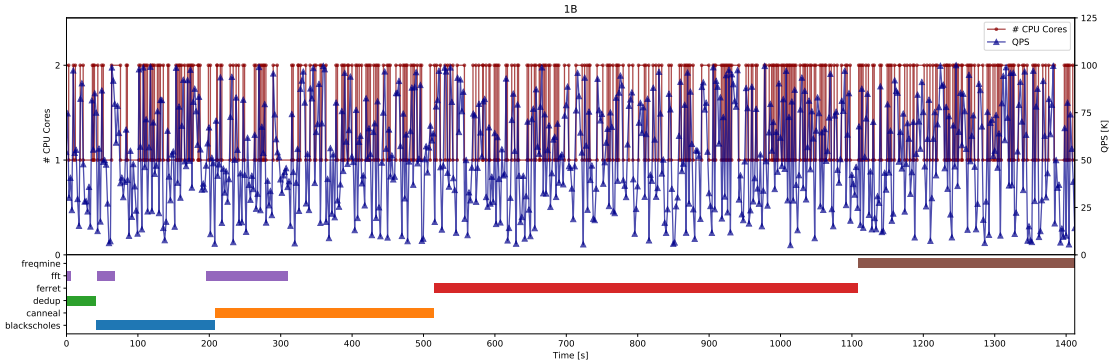
The smallest `qps_interval` is 2 seconds.

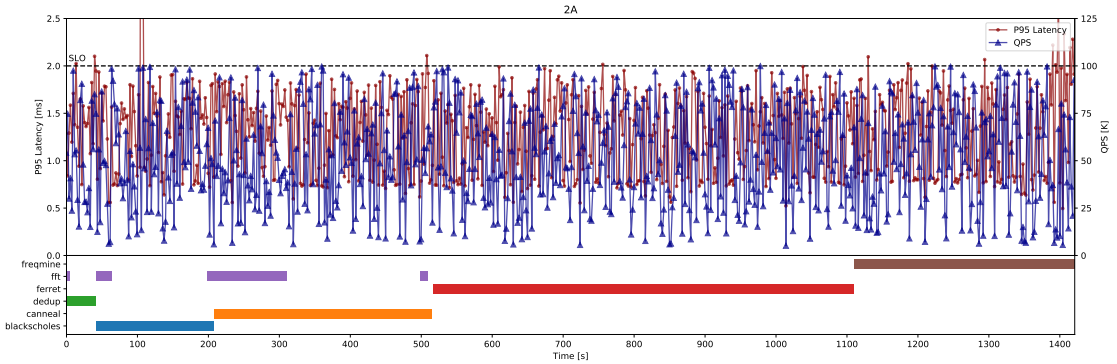
job name	mean time [s]	std [s]
dedup	43.7	3.68
blackscholes	164.1	2.85
ferret	593.8	1.36
freqmine	305.2	5.12
canneal	304.8	2.88
fft	147.4	3.87
total time	1414.0	6.21

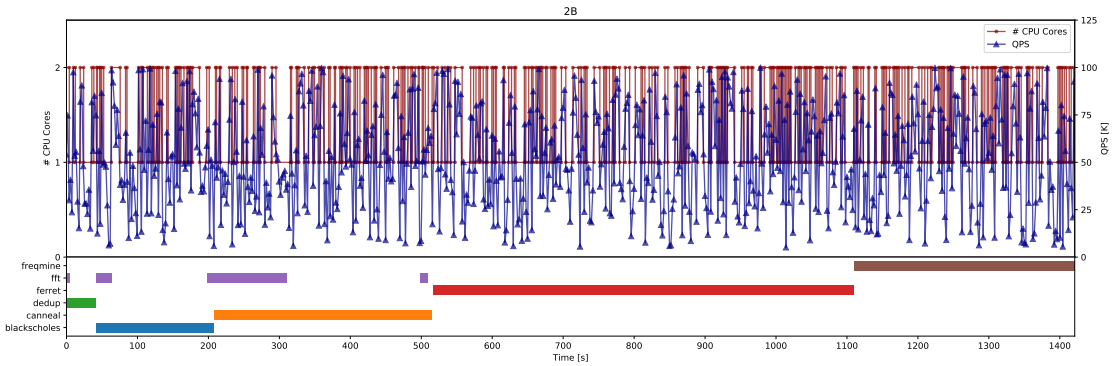
SLO violation ratio:
Run 1: 5 / 707 = 0.7 %

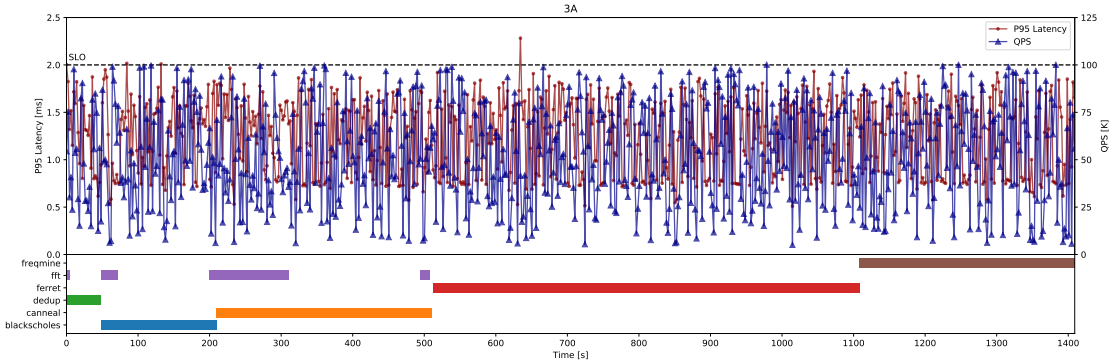
Run 2: 15 / 711 = 2.1 %
Run 3: 4 / 705 = 0.6 %

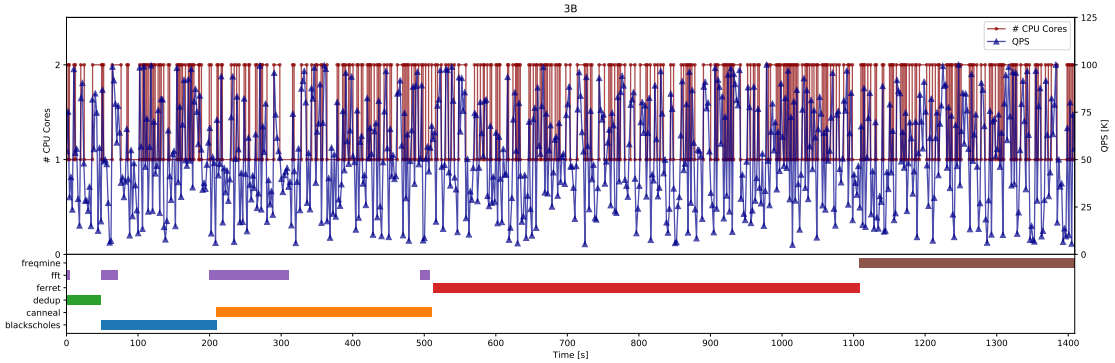









3A
P95 Latency
QPS
SLO
P95 Latency [ms]
QPS [K]
freqmine
fft
ferret
dedup
canneal
blackscholes
Time [s]


3B
CPU Cores
QPS
CPU Cores
QPS [K]
freqmine
fft
ferret
dedup
canneal
blackscholes
Time [s]