Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Cloud Computing Architecture

Semester project report

Group 21
Yiming Wang - 21-959-606
Dominic Steiner - 17-918-988
Adrian Jenny - 18-922-120

Systems Group
Department of Computer Science
ETH Zurich

May 27, 2022

Instructions

- Please do not modify the template, except for putting your solutions, group number, 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.

   For Part 3, you must use the following `mcperf` command:

```
$ ./mcperf -s MEMCACHED_IP --loadonly
$ ./mcperf -s MEMCACHED_IP -a INTERNAL_AGENT_A_IP -a INTERNAL_AGENT_B_IP  \
           --noload -T 6 -C 4 -D 4 -Q 1000 -c 4 -t 20 \
           --scan 30000:30500:10
```

job name	mean time [s]	std [s]
dedup	69.67	0.94
blackscholes	204.67	1.25
ferret	274.33	0.47
freqmine	203.00	0.82
canneal	230.00	1.63
fft	131.33	2.05
total time	281.33	0.47

Answer:
We ran the entire workflow 3 separate times, and measured the start and execution time of each PARSEC job in each run. The latency outputs of memcached given QPS = 30k is collected as well. Results in graphics are shown below.

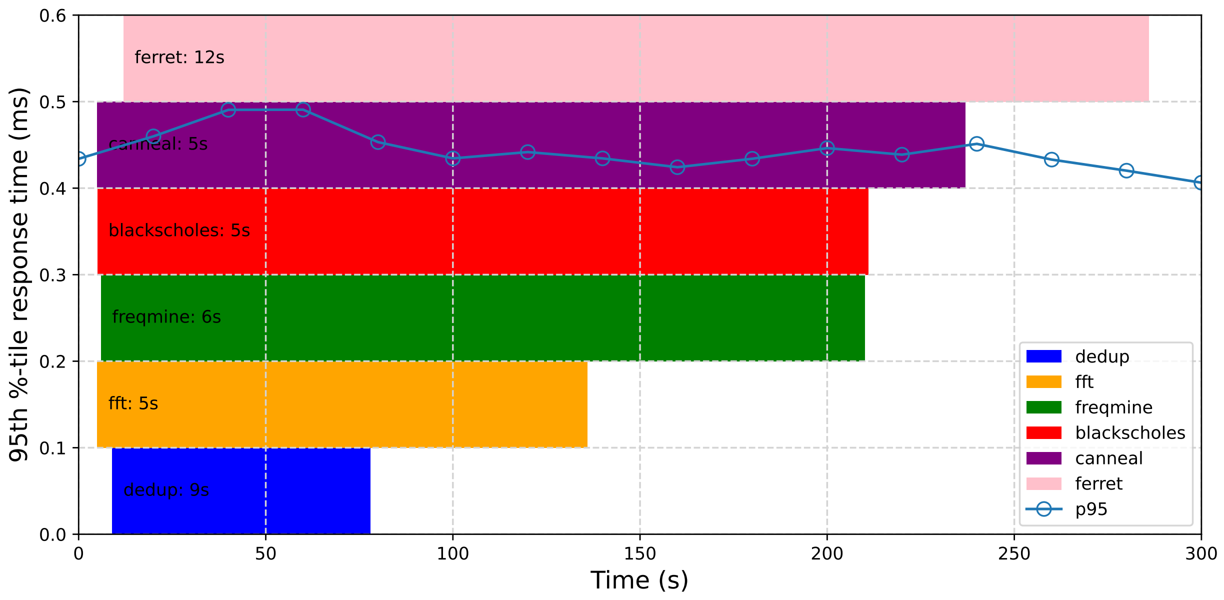

Figure 1: 1. run of the whole workflow

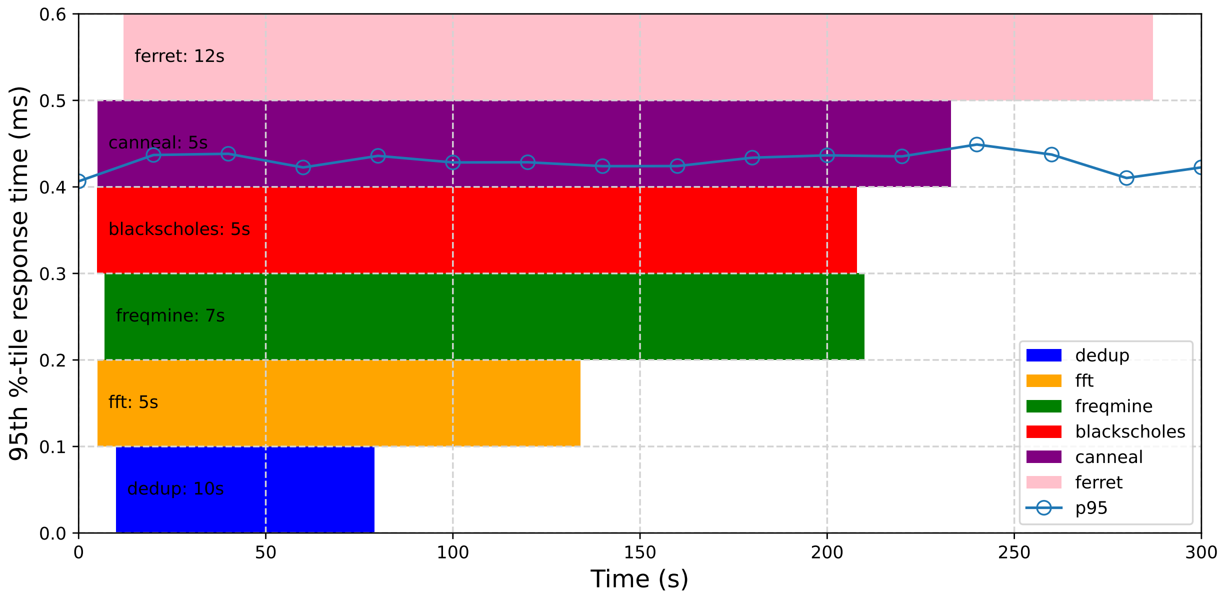

Figure 2: 2. run of the whole workflow

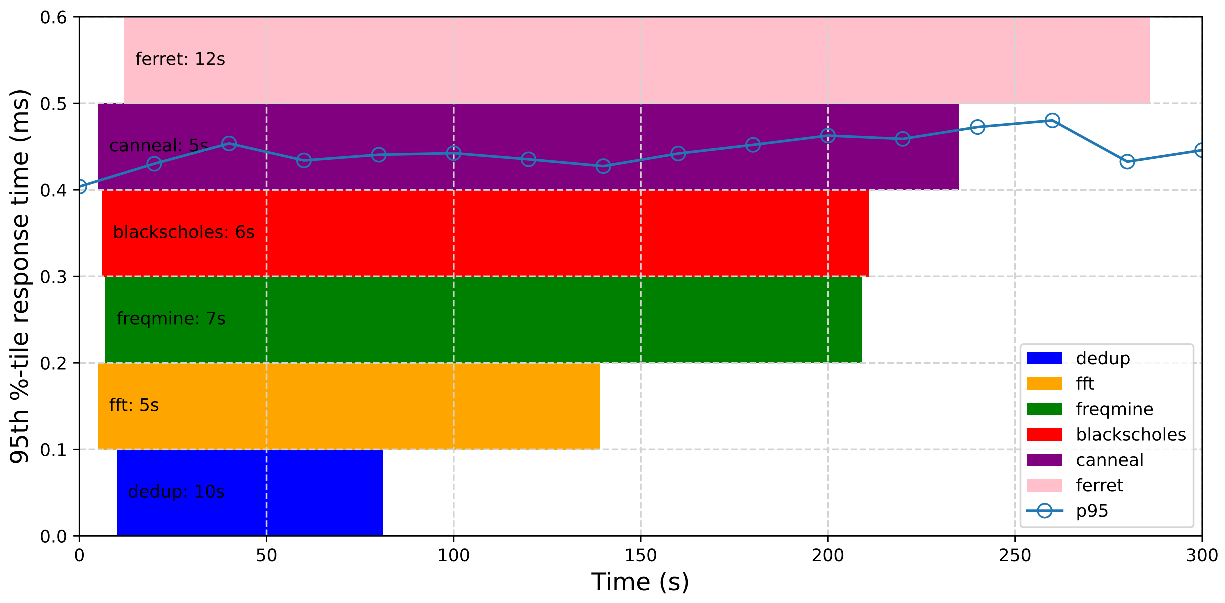

Figure 3: 3. run of the whole workflow

SLO violation ratio:
Due to careful consideration as to how we placed the memcached application and the different PARSEC workloads we managed to stay well below the SLO of 2ms at all three times. In fact, the 95-percentile latency always stayed below 0.5ms. Therefore our SLO violation ratio is 0% for all three runs.

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.**

   Answer:
   We assigned the 6 PARSEC batch apps and memcached on the three given machines, **node-a-2core, node-b-4core, node-c-8core**. We assume that each core on the nodes is homo-

geneous. The exact allocation of PARSEC apps and memcached is shown below.

job name	node	number of threads
canneal	**node-a-2core**	2
dedup	**node-b-4core**	1
fft	**node-b-4core**	1
freqmine	**node-b-4core**	3
blackscholes	**node-c-8core**	2
ferret	**node-c-8core**	5
memcached	**node-c-8core**	1

Specifically we placed memcached on the node-c-8core VM. Several factors and results from previous experiments lead to the decision of this allocation. From part 1 we learned how memcached is particularly susceptible to interference on the CPU or l1 instruction cache and we therefore decided to make sure that memcached could have its own core to avoid these types of interference. From part 2.1 we learned about the influence on the runtime of the PARSEC workloads under different types of interference and also saw how the runtime of most workloads was significantly affected by interference on the CPU and l1 instruction cache. We therefore decided to let most workloads have their own cores, which also minimizes the cost of core scheduling. We also tried to consider interference on the llc as best as possible but thought of this as less of a problem since the llc is significantly larger than higher level caches and we assumed the workloads to not hit this cache as hard compared to when the iBench source of interference was used. Part 2.3 further informed us how a possible speedup would scale when trying to run multiple threads across several cores for a particular workload.

According to the Google Cloud manual (source: https://gcpinstances.doit-intl.com), **node-a-2core** (n2d-highcpu-2) has 2GB memory, **node-b-4core** (n2d-highmem-4) has 32GB memory, and **node-c-8core** (e2-standard-8) has 32GB memory. The PARSEC workload requirements for memory vary across apps. We tested the memory usages of the different workloads privately and found that the fft workload consumed by far the most amount of memory (around 12GB) which already limited the possibilities as to how we could place fft. The other workloads used less then 2GB of memory and could therefore be placed on any machine with regards to memory.

We tried various schedules and forms of implementations, also including trying to use kubernetes' resources.limits.cpu. We also tried to pin the workloads to specific cores directly with the `taskset` command. In the end we noticed that we achieved the best results by not altering these parameters but merely just defining the number of threads via the -n parameter for each workload. With this, the OS' scheduler can take care of placing the threads on the CPU cores. Because we defined exactly as many threads as there were cores (except for node-b-4core). None of these threads appeared to get collocated on the same core and we could therefore avoid interference on the cpu or l1 instruction cache. Our theory as to why this worked better than manually pinning the workloads to a specific CPU is that the OS' scheduler can take much better advantage of its knowledge of the underlying SMT-threads and NUMA architecture and therefore improve memory accesses for memory bound computations. Not defining more threads than there are cores has the additional benefit of avoiding overhead from unnecessary context switches.

We defined one more thread than there are CPU cores on node-b-4core because of the relatively short runtime of dedup on 1 thread. Allocating one core merely for dedup would be wasteful and not fulfilling the objective of trying to utilize as many resources as reasonably possible. Even considering diverse costs of parallelization, synchronization, and context switching across different cores and threads for other workloads, we still identified room to further shorten the total running time by collocating dedup and another PARSEC workload on the same core. This is due to the fact that ferret on node-c-8core still slightly dominates the total runtime.

We implemented this allocation/schedule via two YAML files **memcached_01.yaml** and **schedule_03.yaml**. In the memcached configuration we used resources.cpu.limits:1 from kubernetes. In the schedule file we put all the PARSEC workloads but merely changed the number of threads as an arguments and did not alter any constraints via kubernetes (as described above). In both files we used the nodeselector.cca-project-nodetype to specify on what VM a specific workload had to be placed.

First we started the long-running memcached application via the `kubectl create` command. After starting the memcached measurements we gave the order to start all the PARSEC workloads together by also calling the `kubectl create` command on the schedule file. However, the jobs themselves did not start at exactly the same time. After concluding the experiments we noticed that the workloads roughly started running in the order in which they were listed in the YAML file (barring some minor differences due to setting up the containers etc).This was unfortunate since we had listed the longest running job (ferret) last in the file.
Specifically, across all runs we observed the following orders.

run 1	run 2	run 3
canneal *(tied for first)*	canneal *(tied for first)*	canneal *(tied for first)*
fft *(tied for first)*	fft *(tied for first)*	fft *(tied for first)*
blackscholes *(tied for first)*	blackscholdes *(tied for first)*	blackscholes
freqmine	freqmine	freqmine
dedup	dedup	dedup
ferret	ferret	ferret

With ferret (last) mostly starting about 7 seconds after canneal (first), we noticed that since ferret is the workload with the longest runtime, we could potentially have saved these 7 seconds if we had either listed it first in the YAML file or had tried to explicitly start ferret as the first job. This could have been accomplished by moving ferret to a separate YAML file on which `kubectl create` could be called earlier (by a script with a time offset) than the other workloads. By doing this, the total runtime would get equal to ferret's runtime.

Part 4 [76 points]

1. **[18 points]**

   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
```

   a) 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.

   Answer:
   We ran memcached under the four given configurations for in total three times. The response time is shown below, where the error bars of each data point are depicted by the spacing between the vertical and horizontal lines.

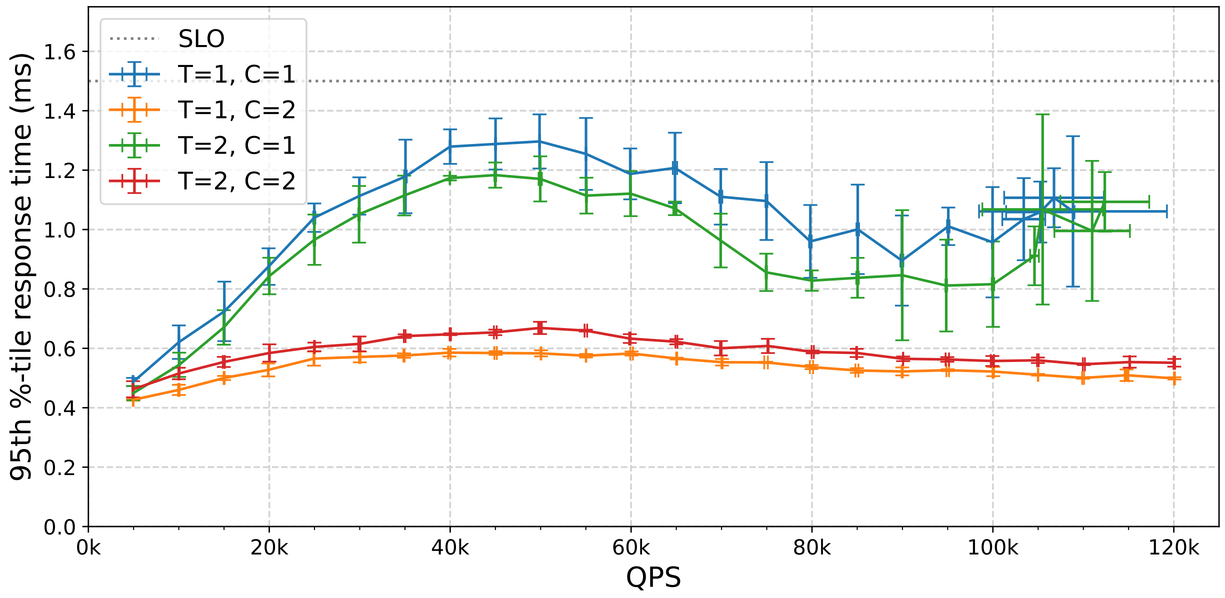

Figure 4: memcached 95%-tile response time given different number of threads and cores

We observed that given one core, on average, having two threads outperformed one thread, as shown by the less 95th%-tile response time. Contrarily, if two cores are available, then having one thread is better than two. These findings were surprising. Our rational as to why one thread on two cores was faster than two threads on two cores is that we think it possible that the two different vCPUs we used in our testing are in fact just two SMT-threads based on the same physical core of the machine. Thus having only one thread could save on communication and context switches compared to having two threads.
Overall, if the number of threads is given, then increasing the cores from one to two leads to a significant performance improvement. In total, memcached reached its optimum performance when using one thread and two cores.

b) To support the highest load in the trace (120K QPS) without violating the 1.5ms 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?
Answer:
We saw in figure 4 that we need two cores in order to even satisfy the requested 120k QPS. This is despite none of the configurations actually violating the SLO for their achieved QPS. We would also choose two threads because we generally think that we'd need two threads for running concurrently on two completely different cores (i.e. not just two different vCPUs on the same physical core as explained in a)).

c) Assume you can change the number of cores allocated to memcached dynamically as the QPS varies from 5K to 120K, 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 1.5ms 95th percentile latency SLO while the load varies between 5K to 120K QPS? i.e., what values of T would you select?
Answer:
As described in b) we already know that we can satisfy most of the requested QPS with one core but we need two cores to satisfy the QPS above about 100k. We would choose two memcached threads to achieve this. Firstly because two threads seem to perform slightly better on one core than one thread and secondly because we think we need two threads to properly utilize two different physical cores (according to the reasoning shown in a) and b)). It also does not make sense to use more than two threads if we are only ever going to run them on a maximum of two cores.

d) Run memcached with the number of threads T that you proposed in c) above and measure performance with $C = 1$ and $C = 2$.

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 120K. In each graph, use two y-axes. Plot the 95th percentile latency on the left y-axis. Draw a dotted horizontal line at the 1.5ms 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.

Answer:

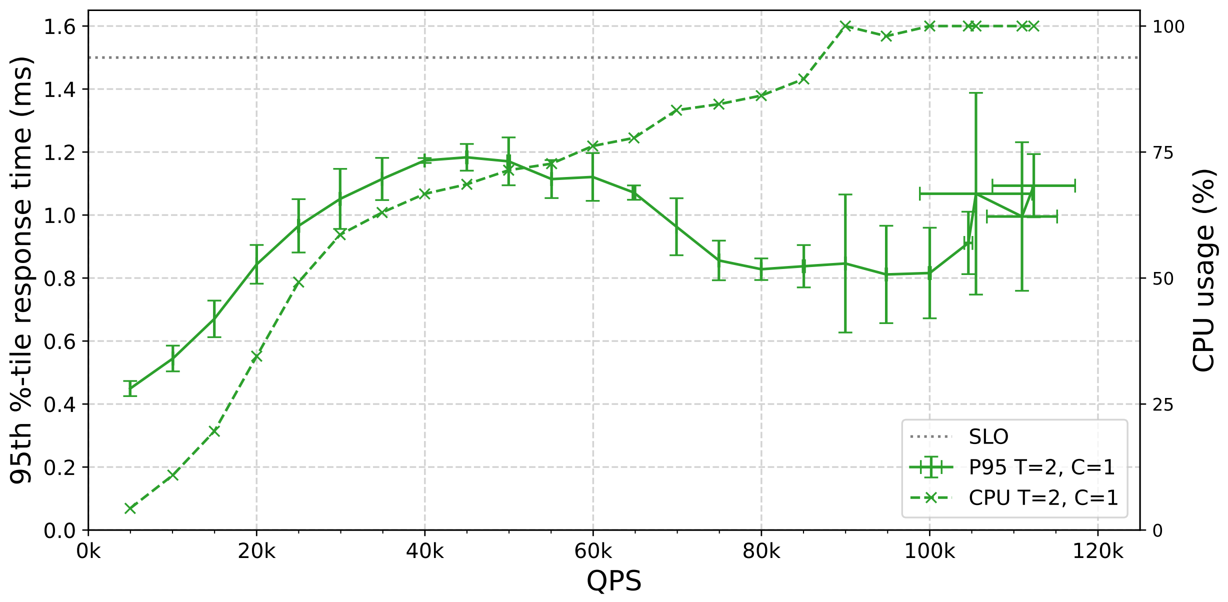

Figure 5: memcached tail latency and CPU usage for 1 core

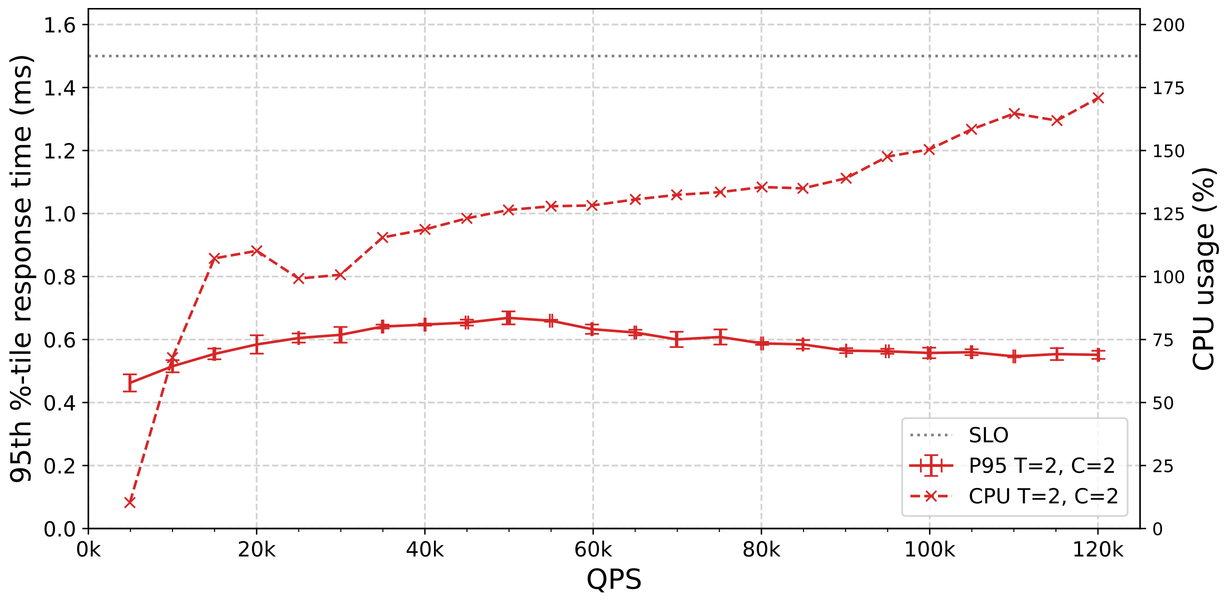

Figure 6: memcached tail latency and CPU usage for 2 cores

2. [**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 1.5ms 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.

Answer:
The decision as to whether memcached should run on one or two CPU cores was purely based on the respective cores' CPU utilization. We start by running memcached on one core at the beginning due to the fact that most of the loads (and therefore probably also the first one) can be satisfied by running on a single core. We switch to two cores whenever the CPU utilization of the single core goes above 95% and we switch back to one core whenever the sum of the CPU utilizations of the two cores goes below 140%. However, we only allow the controller to switch back to one core after it spent 4 seconds with the two cores. This limits the number of core count changes the controller can make. Due to the bursty nature of the different memcached loads we always sample a set of measurements over a short period of time and make our decision based on the largest measurement. This was done to ensure that we do not fall right in between spikes of loads for the decision making. We use CPU utilization because it is (except of course if the utilization is already at 100%) a very good indicator of what loads can be handled well by the CPU. This can be seen in the plots 5 and 6.
We run the PARSEC jobs sequentially in the following order:

(i) ferret
(ii) freqmine
(iii) canneal
(iv) blackscholes
(v) fft
(vi) dedup

Note: The jobs are ordered from longest to shortest but there is no particular reason for this order. Any order should perform about the same because the jobs are executed one after the other.

With exception to fft (which can only run with a number of threads that is a power of two) all workloads run on three threads. This is because we run the workloads sequentially and we want the workloads to be able to leverage the resources of possibly three available CPUs. Also, noted in part 2 of this project that we considered the speedup we can get with three threads to be significant. When memcached runs on one core, the current PARSEC workload can run on the other three cores with one thread each. When memcached runs on two cores, the current PARSEC job only runs on the other two cores. The downside of the second situation is that we might incur some overhead from context switches and unnecessary communication since we have two PARSEC threads running on the same core. But since we spend most of the time with all three cores available, this does not have a significant negative impact. Fft is simply run on two threads. While this is suboptimal in the cases where we'd have three cores available, we still did not use four threads because at least two of them would have ended up on the same core even in the optimal case and thus would have lead to unnecessary overhead. Moreover, we privately tested fft with four threads and four full cores to itself and even in this clean setup the speedup we got with four threads was lower than what we defined as significant in part 2. It is therefore reasonable to assume that the speedup of four threads on only three cores would have been even worse.
While there are some disadvantages to our scheduling policy e.g. how we end up incurring some context switching overhead in the seldom cases where the workloads can only run on two cores, the policy still has many advantages. The simple design leads to less management overhead and we don't need to worry about needing to clean up and finish workloads at the end because we never just pause them. Further, we have the added benefit that we don't need to worry about possible collocation with a different workload that could interfere with the usage of resources that are local to the core. we also have no interference for resources that are used across cores (memBW, llc) except for the necessary interference coming from memcached.
This policy wildly differs from what we did in part 3. Mainly because we had much less resources available. In part 3 we were able to spread the workloads across three machines and run all of them statically in parallel. In this part we chose to run them sequentially and just leverage as much parallelism for a single workload as we could without violating the memcached SLO.
We implemented this policy wholly in python with the help of the docker SDK, psutil and threading modules. Multithreading was necessary because we wanted to be able to sample CPU measurments and schedule PARSEC jobs simultaneously. We used psutil to gather the CPU measurments and also to switch the CPU cores memcached was using. The python docker sdk was used to start the PARSEC workloads and adjust their available cores at

runtime via updating a given container's cpu-set. We chose to run our scheduler on the primary memcached CPU core since we found that it only created negligible interference.

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 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 > 1.5ms, as a fraction of the total number of datapoints.

job name	mean time [s]	std [s]
dedup	43.66s	0.93s
blackscholes	160.42s	4.76s
ferret	528.20s	3.63s
freqmine	321.08s	2.89s
canneal	257.22s	4.24s
fft	125.25s	2.19s
total time	1435.84s	15.63s

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.

Answer:
We had to run all experiments for this tasks on google cloud zone europe-west3-b instead of europe-west3-b because we could not create the cluster in the standard zone. The SLO violation ratios are displayed in the table below. For plots see figures 7, 8, 9, 10, 11, 12.

Note: For all experiments and plots from exercises 4.3 and 4.4 we noticed at the end of our testing that there were seemingly almost no switches to two cores for memcached while the canneal or fft workloads were running. This is even though we still experienced moments

controller run	SLO violation ratio
1	0.69%
2	1.41%
3	0.00%

of high QPS load during these periods. We could not definitely locate the issue after the fact. We believe it possible that the problem could be that memcached could never reach 95% *CPU utilization due to possible interference of these workloads on the shared llc or memory bandwidth. This argument is supported by the relatively similar interference profile for canneal and fft we've shown in part 2 of this project and by the known fact that fft is an inherently memory-bound computation. Assuming this argument to be true, this is an indication that solely relying on the CPU utilization might not be enough information for all scenarios. However, we see that we still managed to satisfy the target QPS at all times.*

4. **[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 > 1.5ms, as a fraction of the total number of datapoints) with the 5-second time interval trace?

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 3.

job name	mean time [s]	std [s]
dedup	48.67s	1.89s
blackscholes	162.39s	2.16s
ferret	543.45s	2.90s
freqmine	325.89s	2.34s
canneal	236.71s	3.61s
fft	130.33s	4.05s
total time	1447.45s	2.32s

Answer:
Note: We had to run all experiments for this tasks on google cloud zone europe-west3-b instead of europe-west3-a because we could not create the cluster in the standard zone.

We did run our controller with the same settings as in 4.3. but reduced the mcperf QPS interval to 5s. Over three runs the average total running time was 27s slower compared to the runs in 4.3. Looking at the data we see that lower QPS interval causes the controller to change the cpu allocations more often. The SLO violation ratios (table 1) did not change significantly compared to 4.3.

controller run	SLO violation ratio
1	0.34%
2	1.71%
3	0.34%

Table 1: SLO violation ratio with 5s QPS interval

Generally we can say, the shorter the QPS interval, the higher the sampling rate of CPU utilization and the associated frequency of decision making whether to increase or decrease the core count would need to be.
Because our controller uses CPU utilisation to make decisions it will always have a slight delay before it can react to such an interval with high QPS. This causes the first few request in an interval to experience higher latency than the later requests. However, as the intervals get shorter, these early request represent a greater share of the overall requests, increasing the 95 percentile latency.
We did try to run our controller with 3s and 2s QPS interval. The controller could not keep up with the 2s interval and we measured a SLO violation ratio of over 8%. So the smallest QPS interval our controller could handle is 3s. The SLO violation ratios are displayed in the table 2. For plots see figures 13, 14, 15, 16, 17, 18.

controller run	SLO violation ratio
1	0.62%
2	0.42%
3	0.21%

Table 2: SLO violation ratio with 3s QPS interval

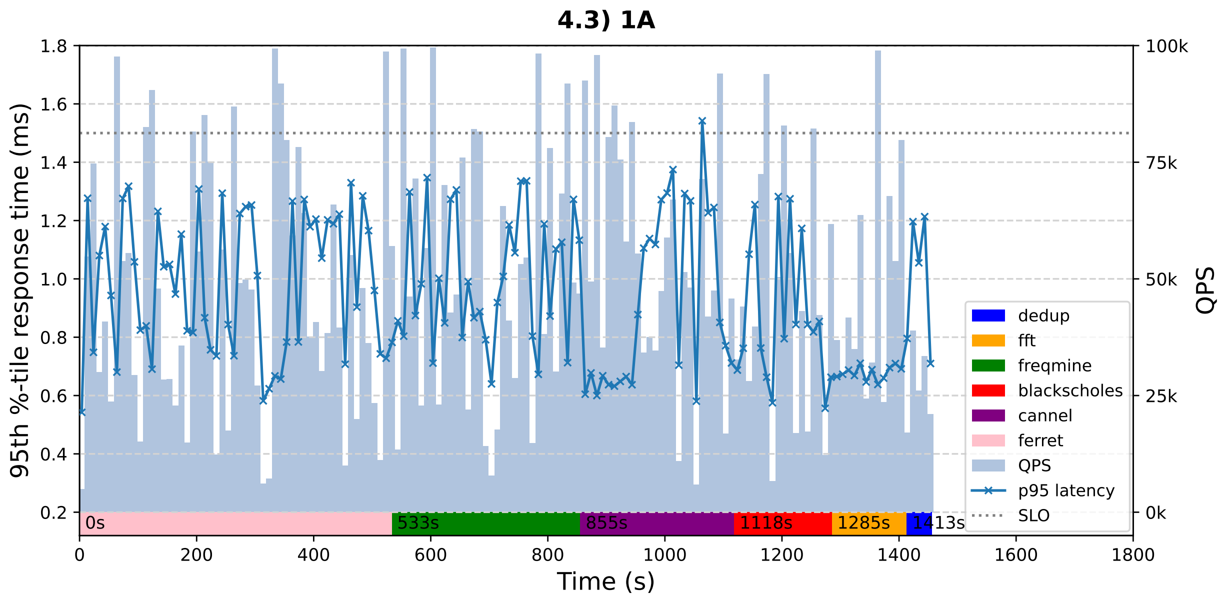

Figure 7: 4.3 1A

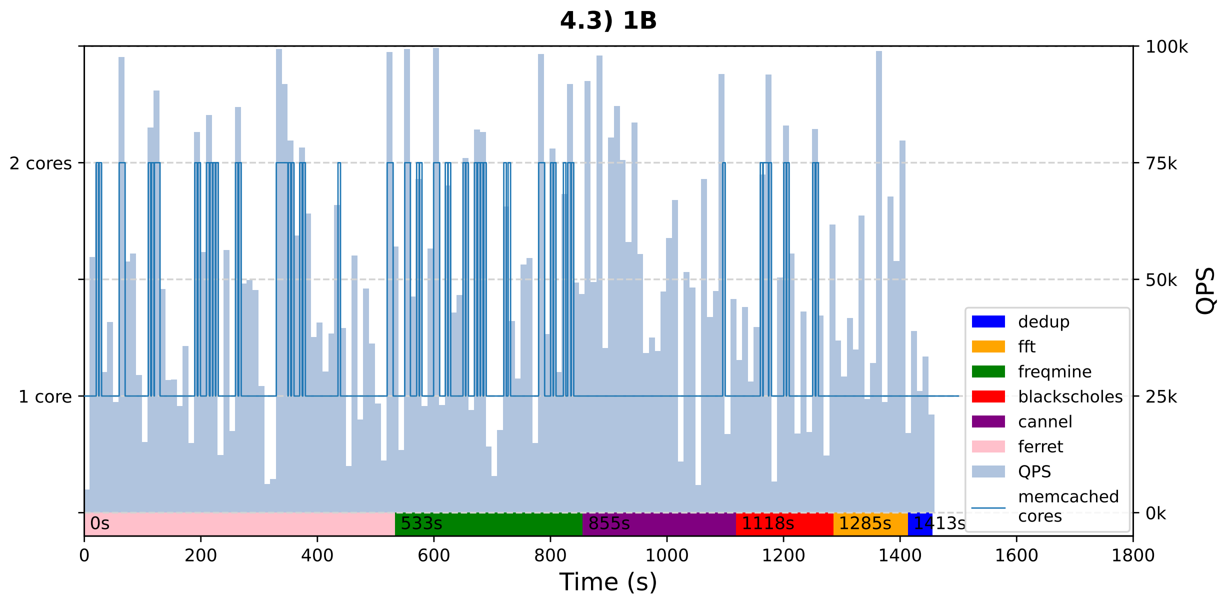

Figure 8: 4.3 1B

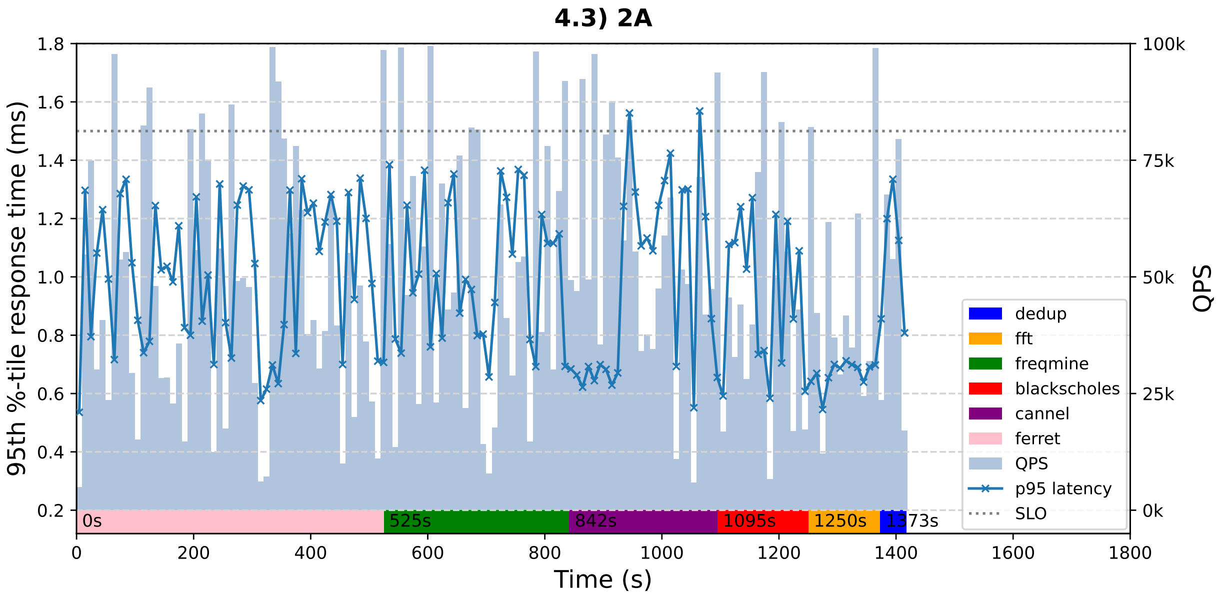

Figure 9: 4.3 2A

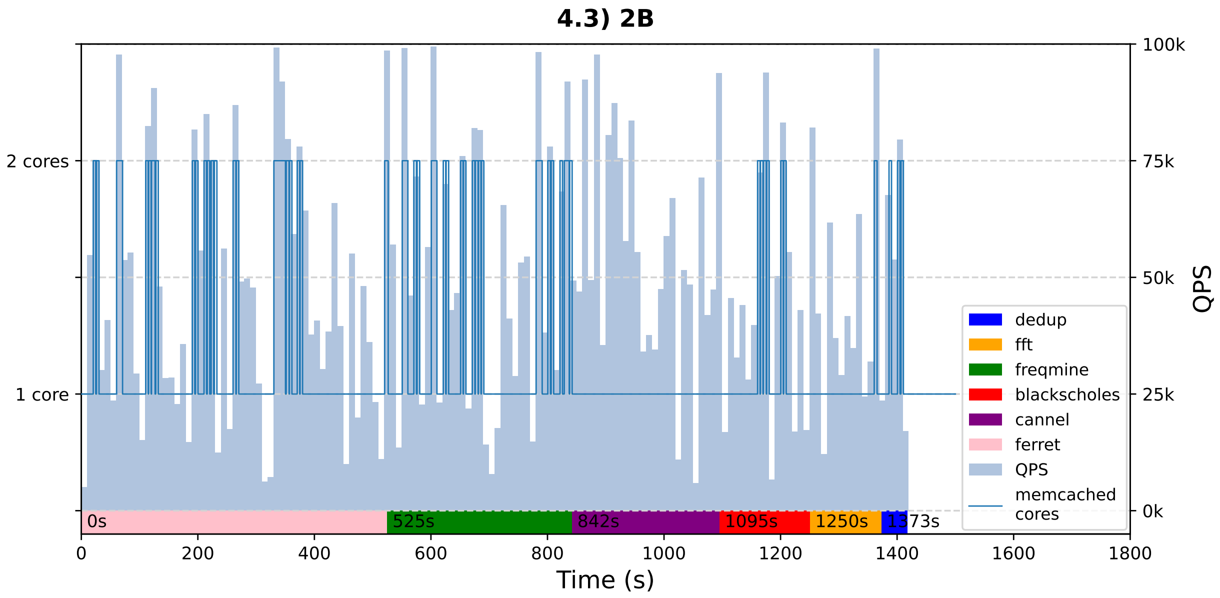

Figure 10: 4.3 2B

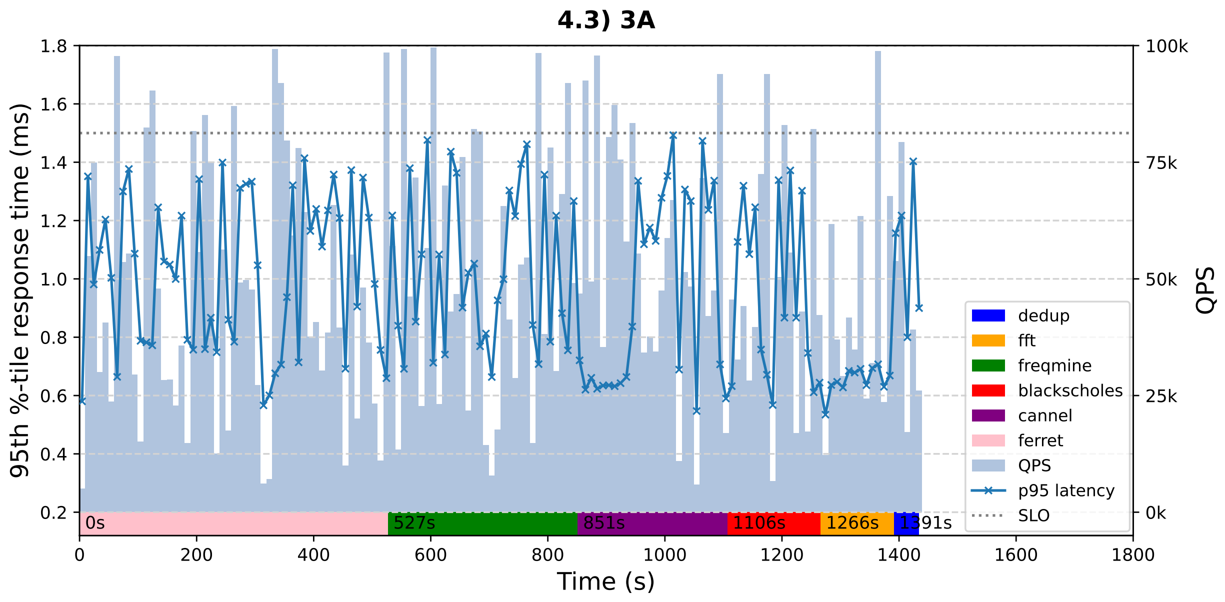

Figure 11: 4.3 3A

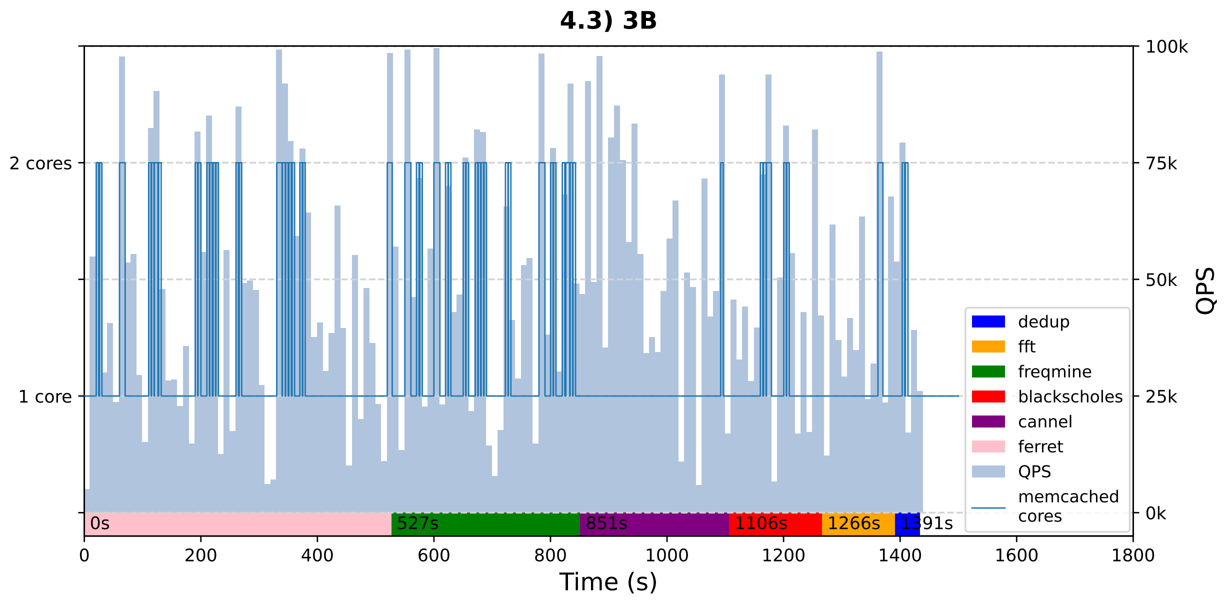

Figure 12: 4.3 3B

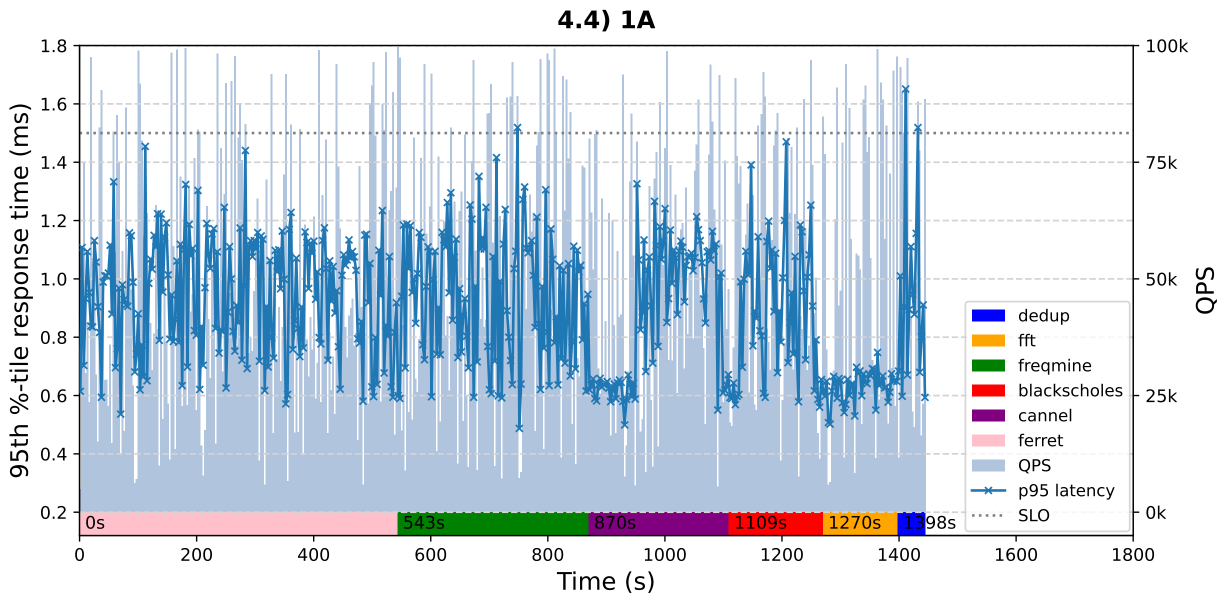

Figure 13: 4.4 1A with 3s QPS interval

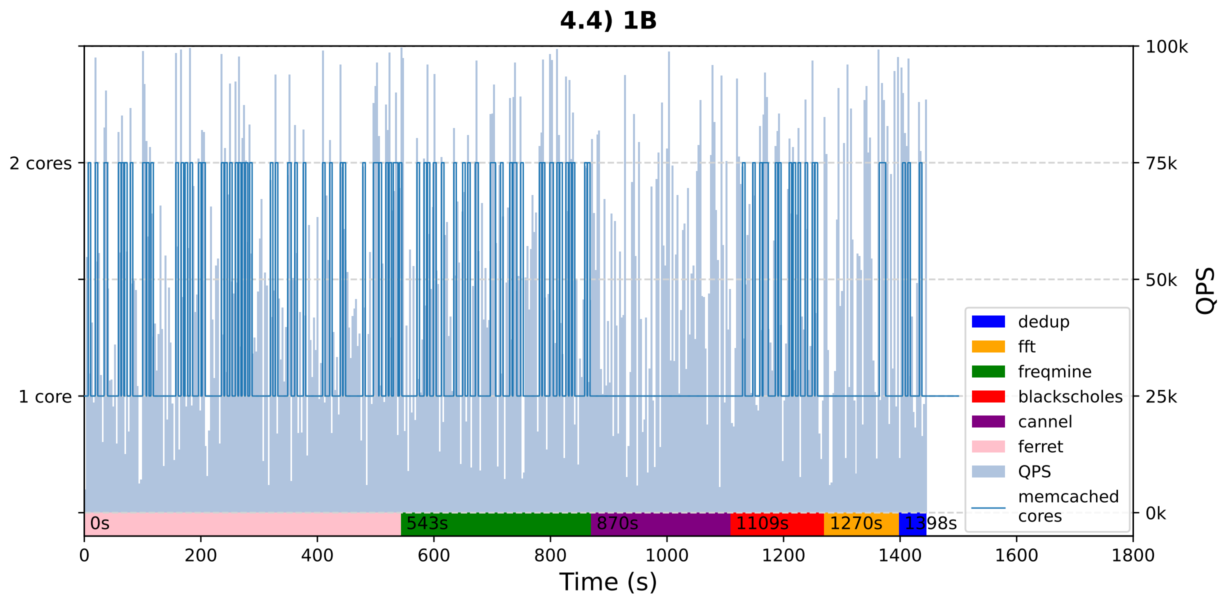

Figure 14: 4.4 1B with 3s QPS interval

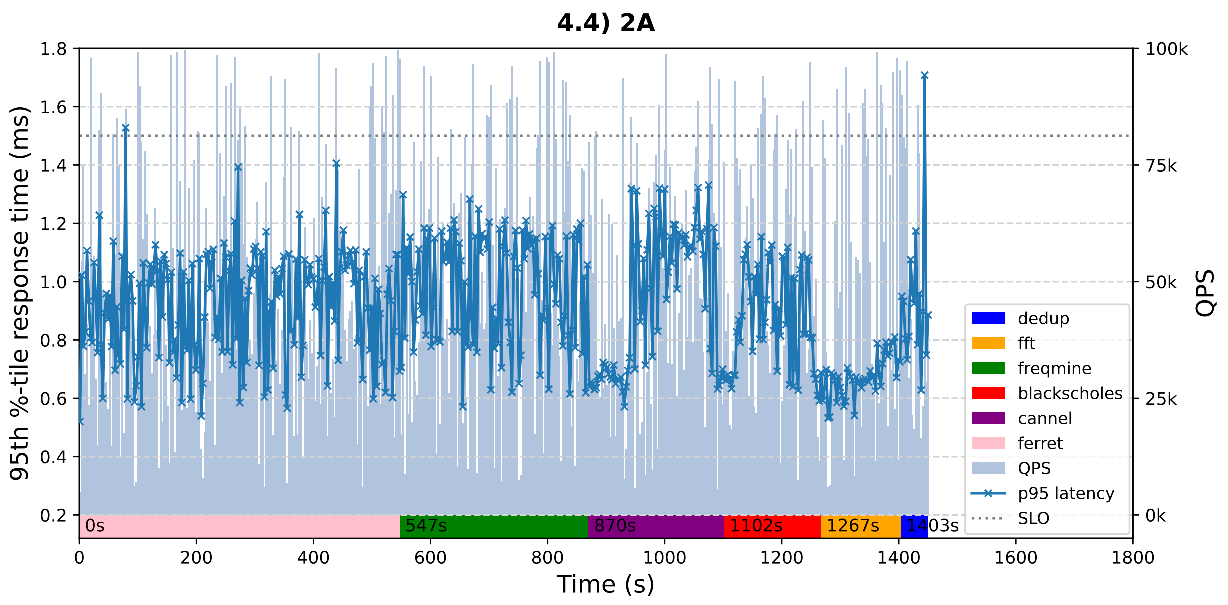

Figure 15: 4.4 2A with 3s QPS interval

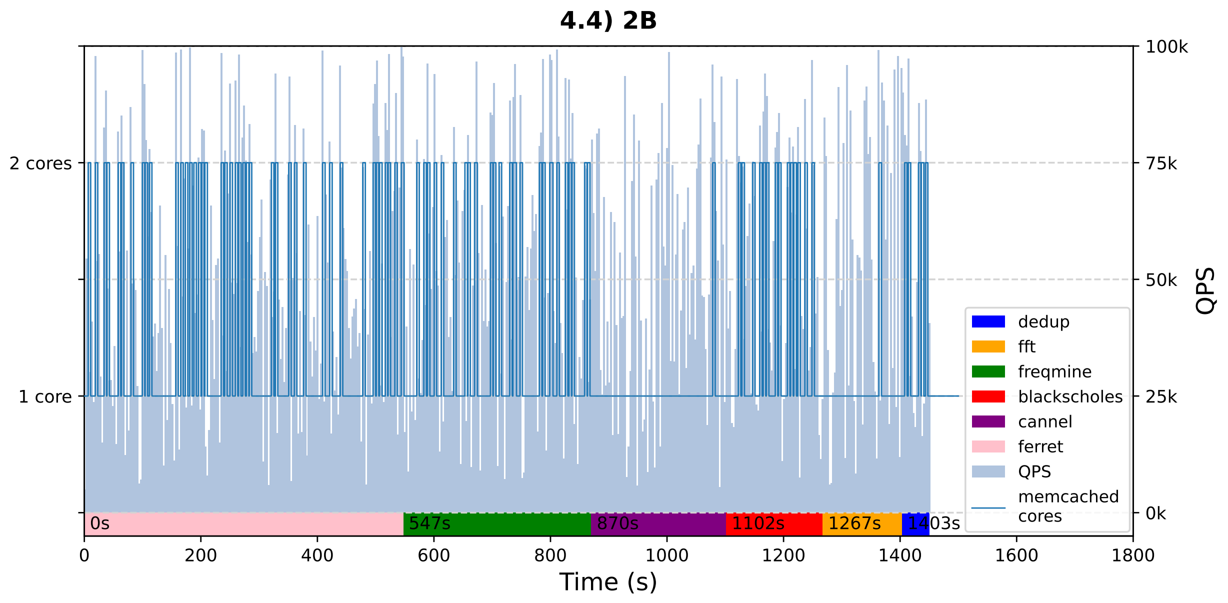

Figure 16: 4.4 2B with 3s QPS interval

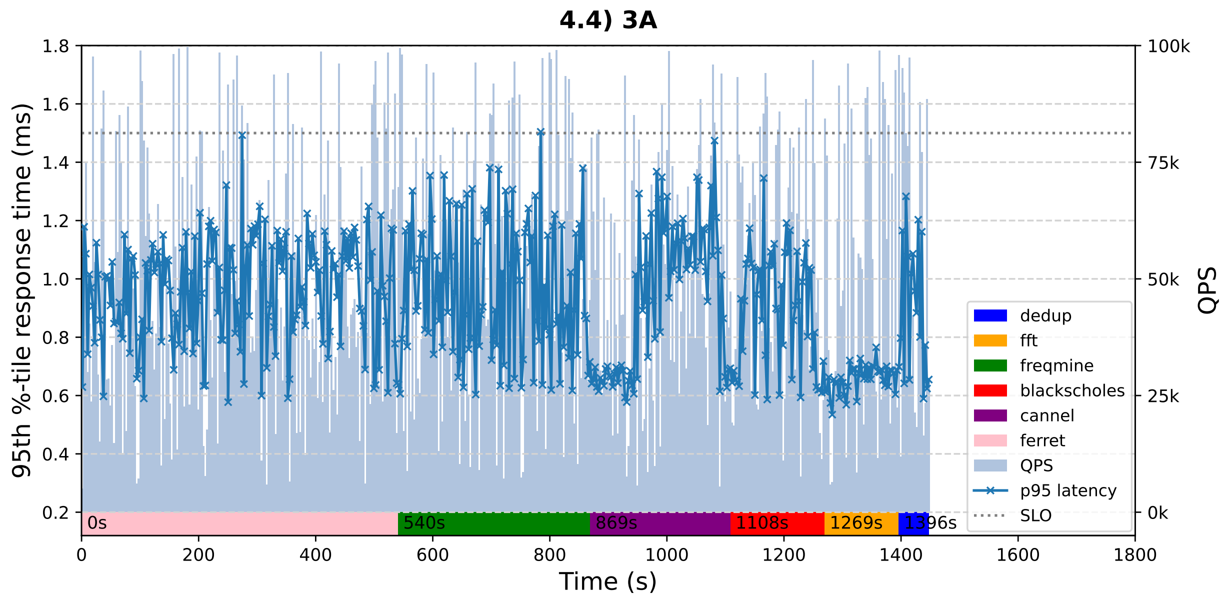

Figure 17: 4.4 3A with 3s QPS interval

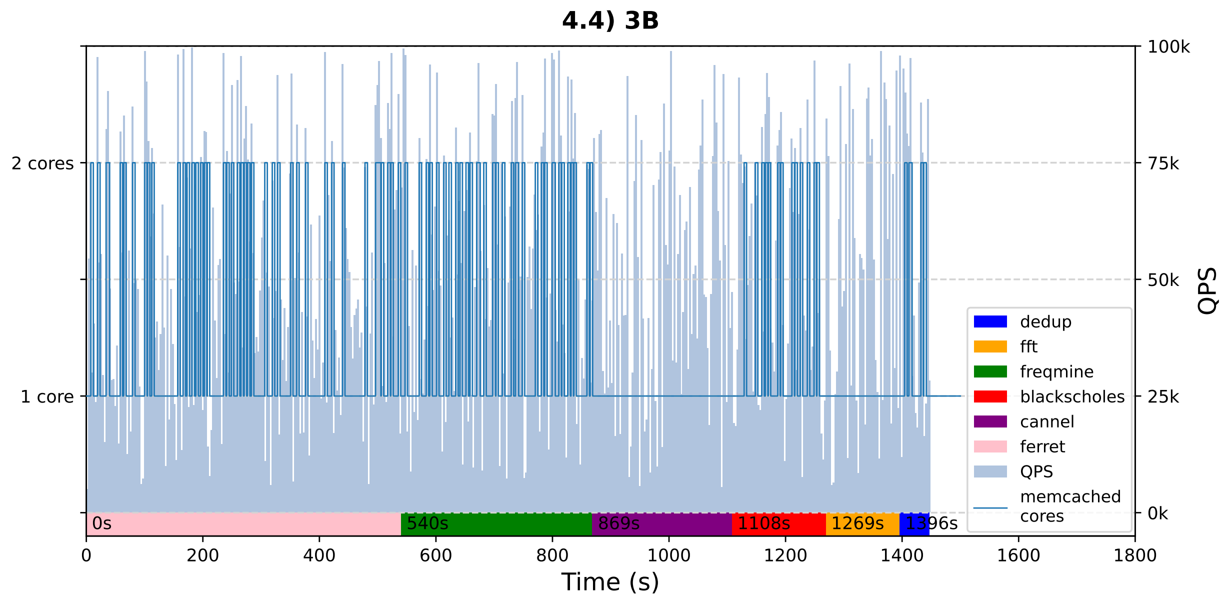

Figure 18: 4.4 3B with 3s QPS interval