Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Cloud Computing Architecture

Semester project report

Group 020
Ambroise Aigueperse - 21-810-692
Témi Messmer - 21-826-870
Monika Multani - 18-061-663

Systems Group
Department of Computer Science
ETH Zurich
May 16, 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	102.67	3.40
canneal	144.67	1.25
dedup	47.33	0.94
ferret	143.67	6.65
freqmine	170.00	0.82
radix	34.00	0.82
vips	114.33	1.25
total time	173.67	2.05

Answer: The SLO violation ratio for memcached is 0% for each run.

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.

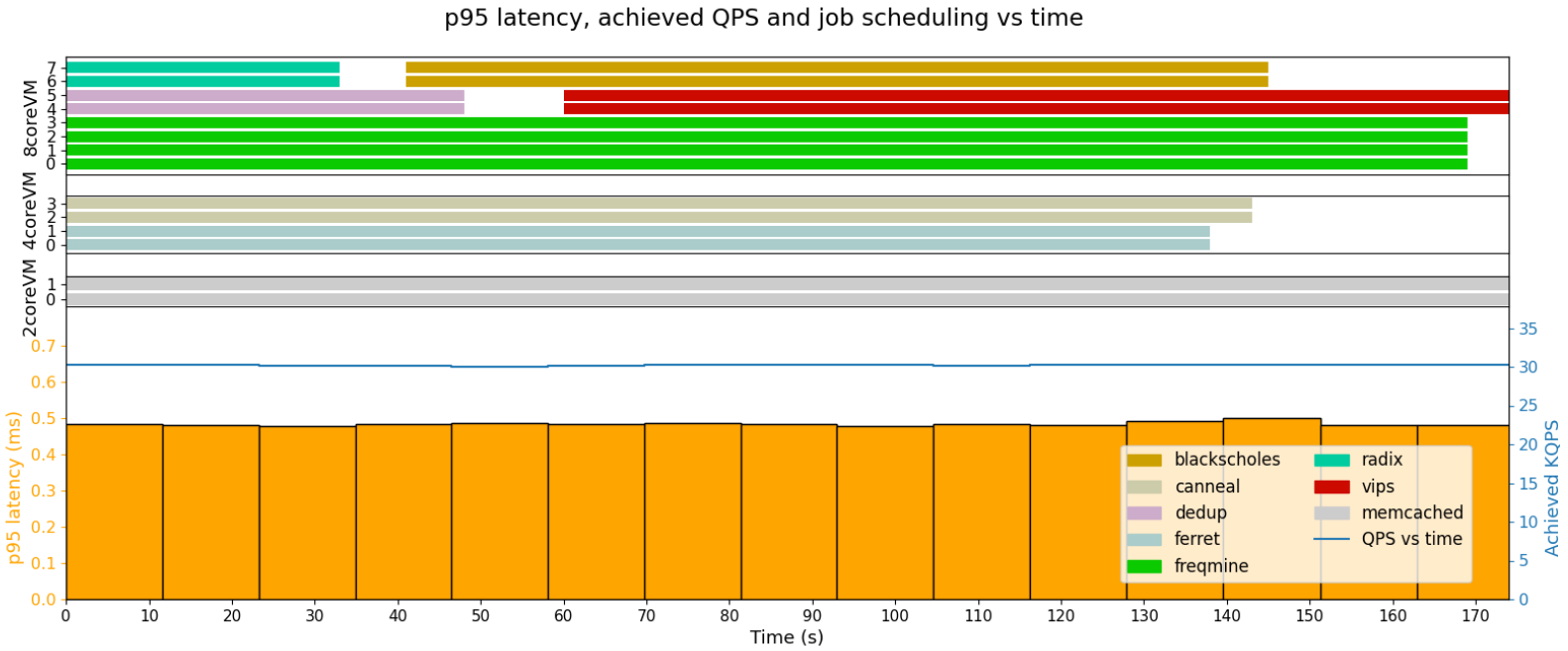

Figure 1: Memcached 95th Percentile Latency Over Time (Run 1)

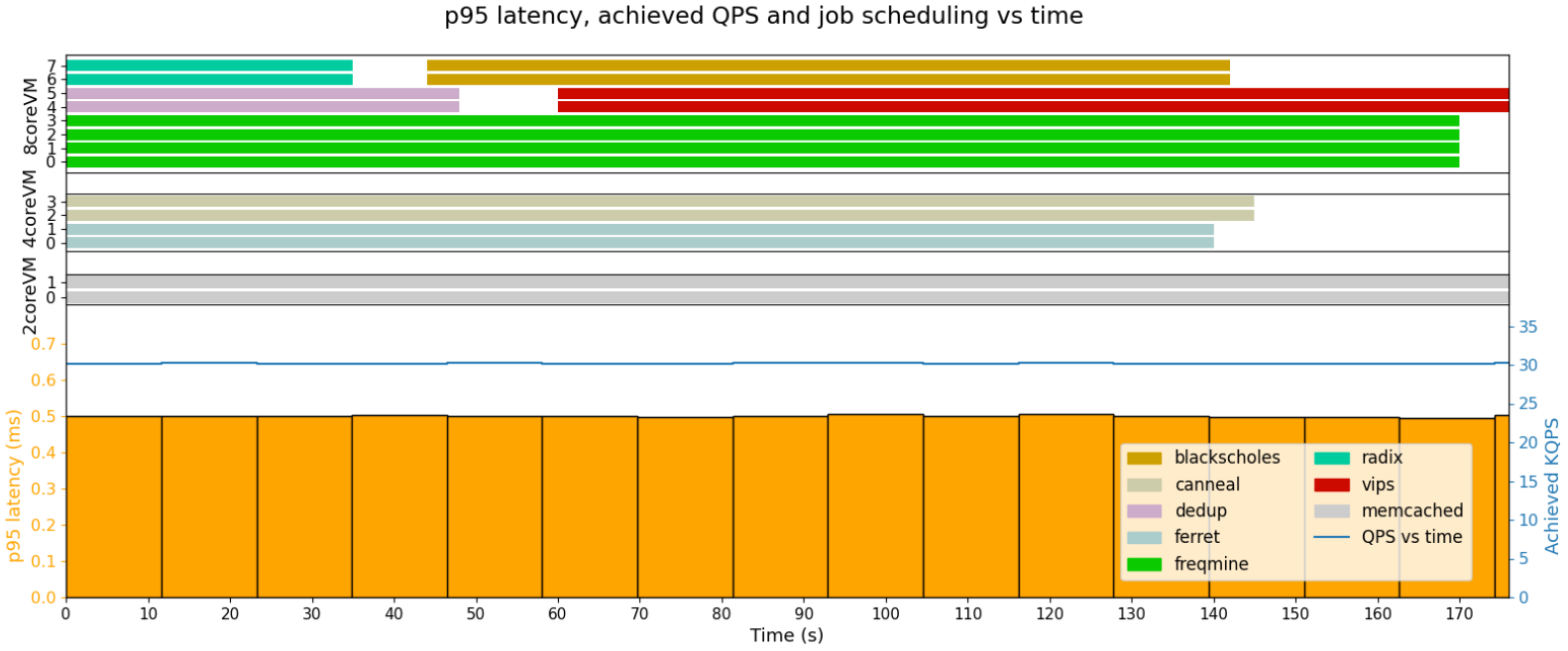

Figure 2: Memcached 95th Percentile Latency Over Time (Run 2)

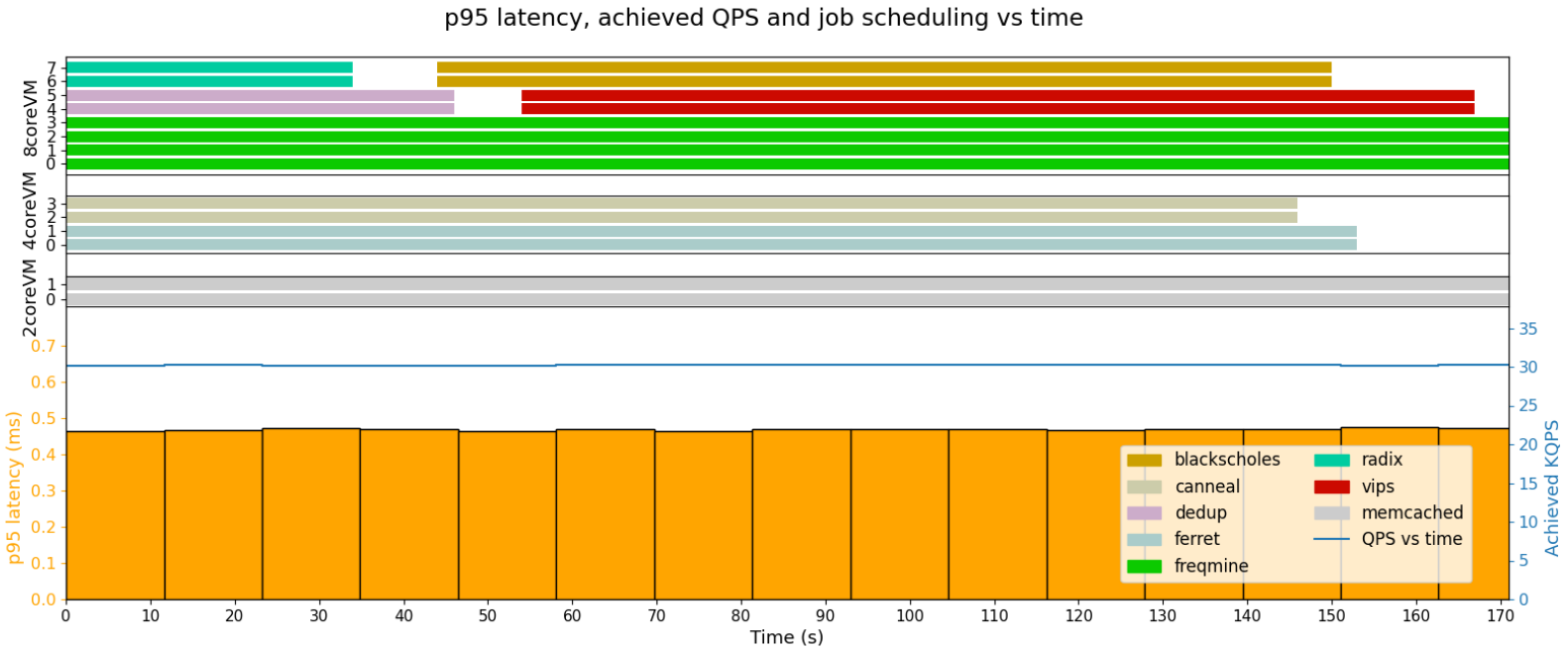

Figure 3: Memcached 95th Percentile Latency Over Time (Run 3)

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

   - Which node does memcached run on? Why?

     Answer:
     We decided that `memcached` should run on the machine labelled *node-a-2core*. This decision was based on the fact that this machine, with its `n2d-highcpu-2` configuration, is optimised for compute-intensive workloads and Part 1 of the project demonstrated that `memcached` uses the CPU extensively. We also decided to run memcached here so that it could be isolated from other workload's interference and that we could guarantee to meet the SLO. Moreover, the 2 cores we could give it by placing it on that machine seemed to be necessary to always meet the SLO.

   - Which node does each of the 7 batch jobs run on? Why?

     Answer:

     – blackscholes: It runs on the machine labelled *node-c-8core* because Part 2 of the project revealed that it has well-balanced resource consumption. Consequently, there was no compelling reason to allocate it to a machine specifically designed for high CPU or high memory consumption, contrary to other workloads for which it is more adapted. Moreover, running it just after radix, which was allocated on *node-c-8core* as explained below, enabled us to have a good resource utilization since the sum of their running time was close to that of the bottleneck which is generally *freqmine*.

     – canneal: We made `canneal` run on the machine labeled *node-b-4core*. We did this because there was no reason to run it on a high CPU machine since part 2 showed us that it had a very low reliance on the CPU. Moreover, we understood in part 2 that canneal is mostly dependent on the LLC cache. Therefore, we thought that when running on a high-memory machine, the reliance on the LLC may be reduced as the penalty for LLC miss (which might happen as `canneal` has a high response to LLC interference and will be collocated with another job) could be reduced since a large RAM might avoid disk access. Besides, high-memory machine typically also have a large LLC, which again fits well with `canneal`'s dependency profile.

     – dedup: We allocated the `dedup` job to the machine labeled *node-c-8core*. This decision was influenced by insights from Part 2, which indicated that `dedup` has high CPU and memory bandwidth dependencies. Consequently, the `e2-standard-8` configuration of *node-c-8core*, which offers a well-balanced mix of CPU and memory resources, seemed most appropriate. Another option could have been to run it on *node-b-4core*; however, it was already hosting `canneal`. We thought about adding both `dedup` and `ferret` on *node-b-4core* as both demand high memory bandwidth, but this would excessively strain the LLC when combined with `canneal`. Therefore, we had to remove either `dedup` or `ferret` and chose `dedup` because it allowed us for better workload combinations (especially when running just before `vips`) on the machine labelled `node-c-8core` which optimised the overall running time.

     – ferret: We allocated the `ferret` job to the machine labeled *node-b-4core*. Part 2 showed us that it had a high reliance on the LLC so the same reasoning as with `canneal` applied to try and reduce this reliance with a high-memory machine. Additionally, `ferret` exhibited the second highest normalized execution time when subjected to memory bandwidth interference. Therefore, running it on *node-b-4core*, a machine with significant memory resources, was deemed appropriate.

- freqmine: We allocated the `freqmine` job to the machine labeled *node-c-8core*. From Part 2, we learned that `freqmine` heavily relies on the CPU and LLI, but less so on memory bandwidth or LLC. This indicated there was no substantial benefit in running it on a high-memory bandwidth machine. Initially, we considered placing it on the high-CPU machine; however, `memcached` was already operational there, and colocating `freqmine` with `memcached` frequently violated the SLO. Consequently, *node-c-8core* was chosen as it offers a high number of CPU cores (four were allocated to `freqmine`), effectively matching `freqmine`'s requirements for CPU intensity and good parallelizability as shown in part 2. This machine's characteristic of having many cores was even more crucial in the fact that `freqmine` appeared to be the bottleneck in our execution and we therefore wanted to reduce as much as possible its execution time by giving it many cores.
- radix: We assigned the `radix` job to the machine labeled *node-c-8core*. This decision was driven by our scheduling strategy, which aimed to collocate the highest number of workloads on this machine, considering its high number of cores. Given its low overall requirements and minimal pressure on shared resources, as shown in part 2, `radix` was well-suited for collocation with multiple workloads. Additionally, its low CPU and memory consumption meant that a standard and balanced machine like *node-c-8core* was a good fit.
- vips: `Vips`, having a profile similar to `blackscholes`, led us to a similar reasoning: it has a well balanced resource consumption, so there was no real need to put it on a machine designed for high CPU or high memory consumption. Moreover, running it just after `dedup`, which was allocated on *node-c-8core* as explained below, enabled us to have a good resource utilization (the sum of their running times was close to that of the bottleneck `freqmine`).

- Which jobs run concurrently / are colocated? Why?

  Answer:

  We will first describe the collocation and concurrency on each individual machine:
  - On *node-a-2core*, there was no colocation as `memcached` ran alone. We attempted to collocate some light workloads like `radix`, but this either broke the SLO too frequently or there weren't enough resources, resulting in one of the workloads aborting.
  - On *node-b-4core*, `ferret` was colocated with `canneal` (for reasons stated previously) and they ran concurrently. This concurrent arrangement was logical as their memory bandwidth reliance, which is a shared ressource, is quite complementary (`ferret` being quite high compared to `canneal` being quite low). Moreover, their respective execution time when running in parallel with 2 cores each was similar, leading to good resource utilization. Also, concurrent execution proved to be faster than sequential, thus shortening the overall execution time.
  - On *node-c-8core*, `dedup`, `vips`, `freqmine`, `radix`, and `blackscholes` are collocated because, as stated in the previous question, this machine was deemed appropriate for each of them and their combination. In terms of concurrency, `freqmine` was identified as the bottleneck, thus it was scheduled to run concurrently with all other workloads, from the start of the experiment to its end. With four remaining workloads to schedule, we considered their reliance on the shared resources, specifically the LLC and memory bandwidth. From Part 2, it was observed that `dedup` places high pressure on these two resources, `radix` exerts very low pressure, and both `vips` and `blackscholes` exert medium pressure. To optimize resource utilization and

minimize the overall execution time, we decided to bind these workloads to run concurrently by pair. Specifically, `dedup` was paired with `radix`, and `vips` was paired with `blackscholes`, balancing the load on the LLC and memory bandwidth. Consequently, `dedup` and `radix` began running concurrently, followed by `blackscholes` and `vips` upon completion. This arrangement of mixing concurrent and sequential scheduling optimized resource utilization, as the combined execution times of `radix` and `blackscholes` closely matched those of `dedup` and `vips`.

Finally, to summarize, regarding overall concurrency across machines: `memcached`, `ferret`, `canneal`, `dedup`, `freqmine` and `radix` initially run concurrently. Once `dedup` and `radix` complete their executions, `vips` and `blackscholes` commence running concurrently. This concurrency pattern was chosen to minimize the total execution time and maximize resource utilization, ensuring that each CPU core and other resources are kept as busy as possible.

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

  Order (to be sorted): canneal (1 ex aequo), dedup (1 ex aequo), ferret (1 ex aequo), freqmine (1 ex aequo), radix (1 ex aequo), blackscholes (2), vips (3)

  (this order specifies the order in which the workloads are started)

  Why:

  As previously explained, the only hard ordering dependency we established is that `vips` runs after `dedup` and `blackscholes` runs after `radix`, ensuring that `dedup` and `radix` are concurrent, followed by concurrent execution of `blackscholes` and `vips`. This order could have been reversed (`vips` → `dedup` and `blackscholes` → `radix`) without significantly altering the overall execution time or resource utilization. For other workloads, apart from `blackscholes` and `vips`, launching all at the beginning proved to be the most efficient strategy to minimize the overall execution time, avoiding the introduction of additional dependencies. Indeed, the execution times given our scheduling are closely aligned: `ferret` is comparable to `canneal`, which is similar to the sequential execution of `radix` and `blackscholes`, which in turn is similar to the sequential execution of `dedup` and `vips`, closely matching the execution time of `freqmine`, which is the bottleneck. As we weren't able to further reduce the execution time of bottleneck `freqmine` without impacting too much the already good overall execution time or the ressource utilization, we maintained the described schedule.

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

  Answer:

  A first thing to note here, which is true for each workload is that we allocated the same number of threads to a process as the number of cores it could utilize. The reason behind this choice is that allocating one thread per core ensures each core is fully utilized, maximizing computational efficiency and throughput. Indeed, each tasks being divided across all available cores reduces idle time and enhances resource utilization for compute-intensive workloads, compared to having less than 1 thread per core. Moreover, it proved also to be better for the performance, since it avoids context switching or thread switching overhead and potential resource contention that would happen in case there were more than 1 thread per core.

  – blackscholes:

    For `blackscholes`, `dedup`, `vips`, `freqmine`, and `radix`, they all run on the same machine, so a first thing to explain is how we decided the allocation of cores (as

it is linked to the number of threads as previously explained). As detailed in the previous question, `vips` is scheduled to run after `dedup` finishes, making it logical to assign them to the same set of cores. Similarly, `radix` and `blackscholes` share another set of cores. Consequently, we divided the 8 cores of this machine into three groups: one set for `radix` and `blackscholes`, one for `dedup` and `vips`, and one for `freqmine`.

From our observations in Part 2(b), we noted that the most beneficial speedup in jobs' execution relative to an increase in the number of threads occurs when transitioning from one thread to two. This finding led us to want to allocate at least two threads—and therefore two cores—to each job. During scheduling tests, `freqmine` was consistently identified as the bottleneck, prompting us to allocate as many cores as possible to it. Consequently, we distributed the cores as follows: two cores were assigned to `dedup` and `vips` to run sequentially, four cores to `freqmine` running alone, and two cores for `radix` and `blackscholes` to also run sequentially. With this core distribution in mind and adhering to the established relationship between the number of cores and threads, `blackscholes` operates with two threads.

- canneal: 2 threads were allocated to `canneal`. The explanation can be found below at `ferret`, since we reasoned about their thread allocation together considering their collocation on `node-b-4core`.
- dedup: 2 threads were given to `dedup` as explained in the `blacksholes` solution.
- ferret: As explained in the previous question, `ferret` is running concurrently with `canneal` on *node-b-4core*. Similar to the approach used for `blackscholes`, we aimed to allocate at least two threads per workload for both `Ferret` and `canneal`, as part 2 showed that the optimal ratio of speedup increase versus the number of thread increase occurs when moving from one thread to two threads. The only possibility if we wanted to run the jobs concurrently was therefore to give them 2 cores and 2 threads each.
- freqmine: 4 threads were given to `freqmine` as explained in the `blackscholes` solution.
- radix: 2 threads were given to `radix` as explained in the `blackscholes` solution.
- vips: 2 threads were given to `vips` as explained in the `blackscholes` solution.

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

  Answer:
  The first files that we modified were the YAML files for all the workloads, including `memcached`. In these files, we utilized the Kubernetes feature that allows us to specify which core a workload can run on, employing the `taskset` command in the command arguments. Furthermore, we controlled the number of threads using the `-n` argument. We also used the Kubernetes `nodeSelector` feature to designate on which machine each workload should run by specifying the label of a node. Finally, it was unnecessary to write resource requests or limits in the YAML files, as the sets of CPU cores allocated to each workload did not intersect for most; and for those that did intersect, they ran sequentially, thereby eliminating any competition for core resources among the workloads.
  Apart from the YAML files, we tried to automate the testing of our designed scheduler. To this end, we created a bash script named `part3_one_run.sh`. This script initiates by creating the cluster and installing the necessary dependencies on the machines by

sending bash files to each machine containing the appropriate commands to execute, before actually executing those files. Once this setup is done, the jobs are scheduled. For the scheduling, `freqmine`, `radix`, `dedup`, `ferret`, and `canneal` are launched immediately after `memcached`'s workload is correctly set up. Then, for `blackscholes` and `vips`, we poll the cluster using `kubectl get jobs` every second to check if `dedup` or `radix` are completed. Upon completion, we trigger the next appropriate workload. Once all the workloads have finished, which we check during our polling, we extract the JSON file that summarizes the runs. This file is then passed as an argument to `get_time.py` to analyze the execution metrics.

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

  Answer:
  The design choices are already delineated above. To summarize the trade-offs discussed in various questions, a first trade off included balancing the SLO, preventing job failures due to resource shortages, and achieving a reasonable execution time. This trade-off was particularly evident when attempting to collocate `memcached` with other workloads, which often resulted in high variability in `memcached`'s latency and occasionally breached the SLO, sometimes even causing the collocated jobs to crash. This situation necessitated the isolation of `memcached` on *node-a-2core*, despite the potential benefits of shared resource utilization for overall execution time. Another significant trade-off involved balancing resource utilization against the total execution time. As previously explained, our strategy was to make the total execution times across our defined different sets of cores as close as possible, which we think we achieved relatively well. While it might have been possible to further optimize the percentage of resource utilization throughout the whole run, our testing indicated that additional gains would likely incur a substantial increase in overall execution time—a cost we deemed too high.

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.

Plots:

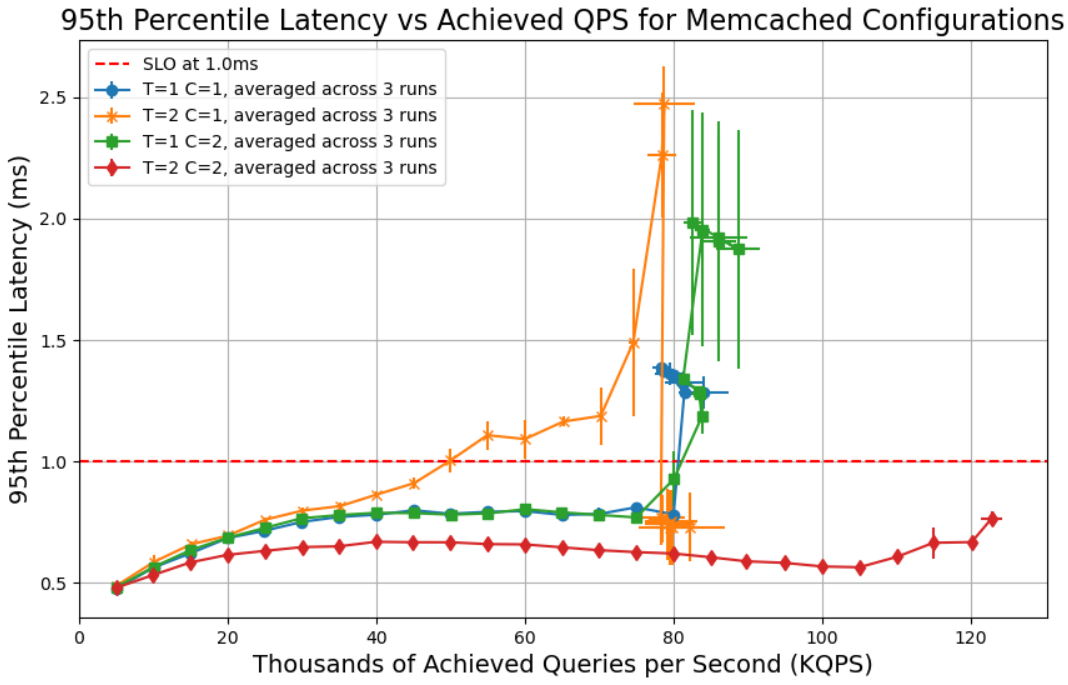

Figure 4: 95th Percentile Latency vs QPS using different configurations for memecached.

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:
Only with two cores and two threads we are able to match the target QPS and SLO. Having 1 core and 1 thread or 1 thread and 2 core leads to quite similar results, as expected because the thread will only be able to utilize effectively one core, but we can't reach over 85K achieved QPS and when exceeding 81K achieved QPS, the meeting of the SLO is compromised. Finally, having more threads than cores (i.e. C=1 and T=2) leads to the worst results, as expected because of the context switch overheads that occur on the core where threads compete.

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:
It will need 2 cores and 2 threads, since as explained before, the other configurations can't even achieve more than 85K QPS.

c) [1 point] Assume you can change the number of cores allocated to memcached dynamically as the QPS varies from 5K to 125K, but the number of threads is fixed when you launch the memcached job. How many memcached threads (T) do you propose to use to guarantee the 1ms 95th percentile latency SLO while the load varies between 5K to 125K QPS?

Answer:
2 threads should be chosen, since with 2 threads, we will be able to meet the SLO and target QPS over 90K QPS when giving more than 1 core to memcached. Having more threads is not necessary and might even be harmful as it could potentially lead to resource contention issues when the core count is limited (there could indeed be the same effect as that illustrated with C=1 and T=2 in the preivous graph). Moreover, if only 1 thread was given to memcached, it would fail to reach objectives with any given number of cores, especially when experiencing a high target QSP, as observed with our measurements. Yet, it has to be noted that having 2 threads, even if the best solution, is not perfect since it will struggle when only 1 core will be given to memcached, so the design of our scheduler should take this into account.

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:

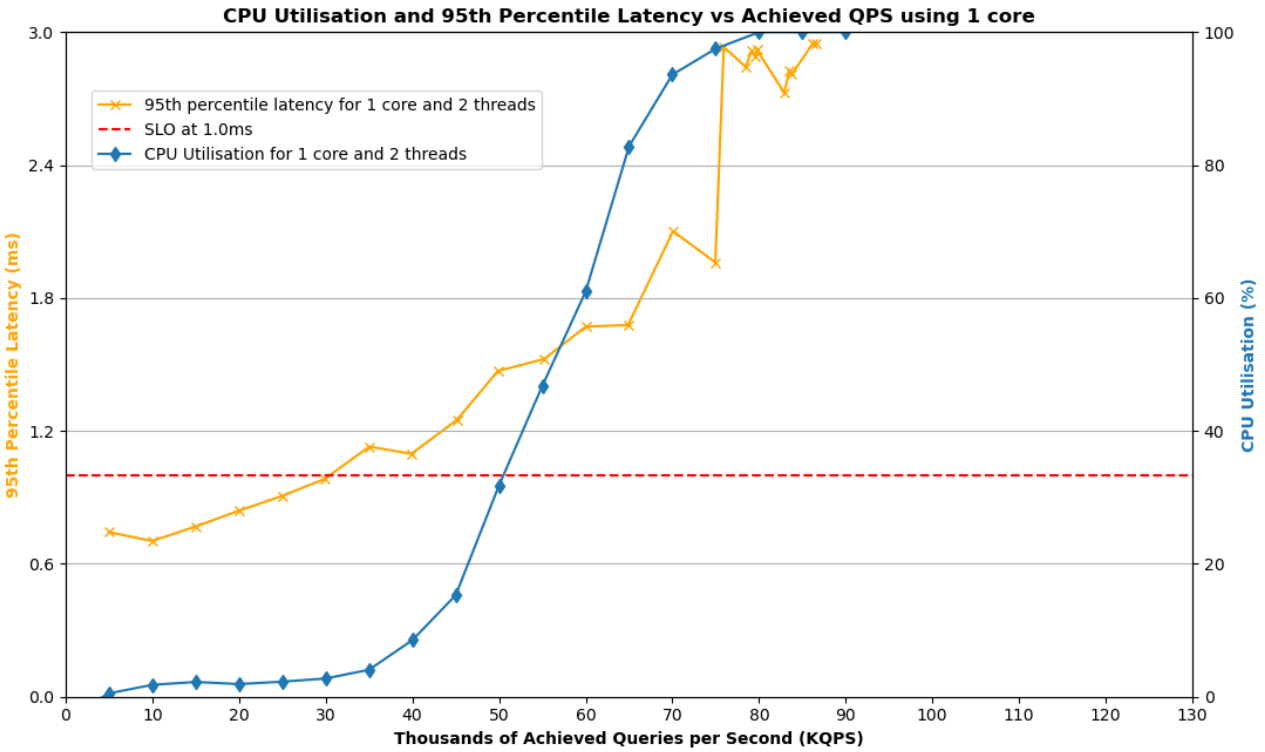

Figure 5: CPU utilisation and 95th Percentile latency vs QPS using 1 core and 2 threads

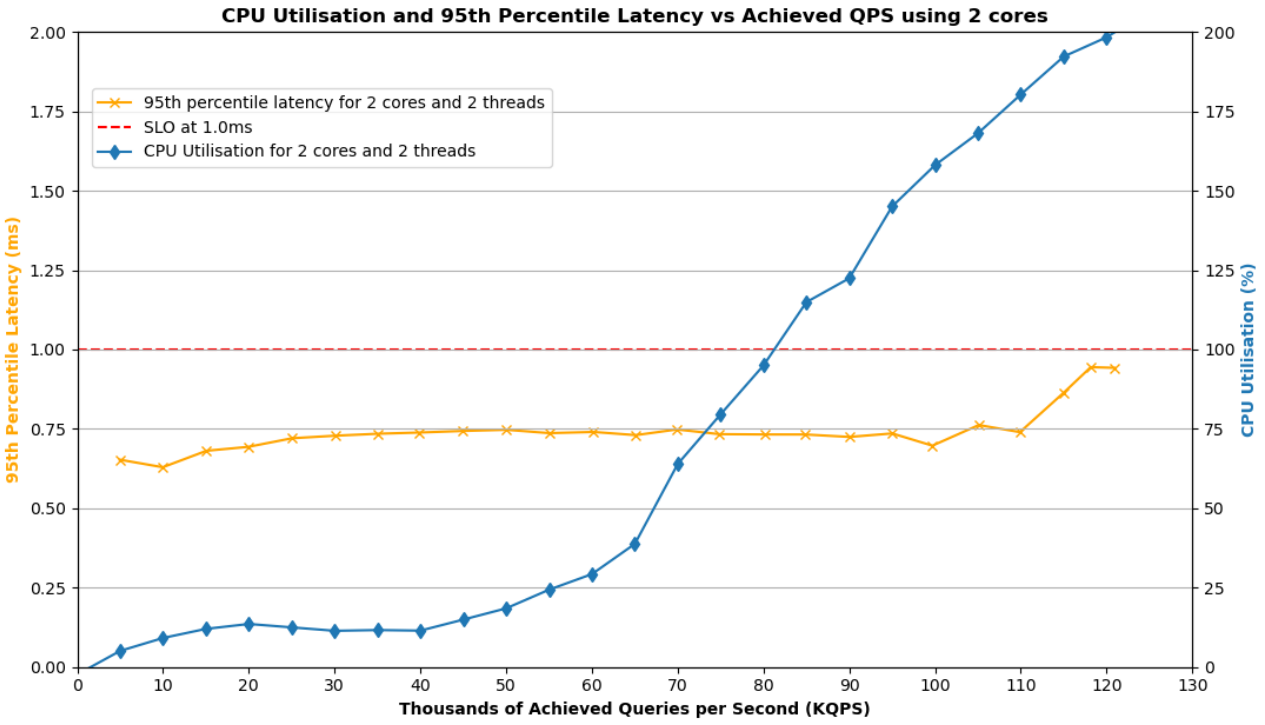

Figure 6: CPU utilisation and 95th Percentile latency vs QPS using 2 cores and 2 threads

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 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:
  Our scheduling policy assigns 2 threads to memcached, dynamically adjusting its core allocation between 1 or 2 based on CPU utilization thresholds to meet the SLO with minimal resource usage. Consequently, at any given time, there are 2 or 3 cores left available for the PARSEC jobs. We hence decided to have 2 queues on which the jobs are statically distributed, q2 and q3 where jobs have respectively 2 and 3 threads. Q2 is scheduled when 2 cores are available for the PARSEC jobs and q3 is scheduled when 3 are available. Jobs within each queue are processed sequentially in general, as it gives them enough resource and enables a good utilization, since they generally have a matching number of threads and cores given. The only case when they can run in parallel within a queue is when q3 empties before q2 and 3 cores are available for the PARSEC jobs.
- How do you decide how many cores to dynamically assign to memcached? Why?
  Answer:
  Memcached dynamically alternates between one and two cores, depending on the workload. Initially, we assign one core to memcached and monitor its CPU utilization every 0.24 second. If memcached's core CPU utilization reaches a predefined threshold t1, we then allocate two cores to memcached. We continue to monitor every 0.24 seconds, and if the combined CPU utilization of memcached's cores falls below a second threshold t2, we revert to a single core allocation. This transition of core allocation goes on until the end of the scheduling. To explain why we chose to dynamically adjust the core allocation like that, question 1 of part 4 showed that, with 2 threads, 1 core allocated to memcached

is sufficient for small loads while 2 are needed for high loads. Therefore, always giving 1 core to memcached would have broken the SLO for high loads and conversely, giving more than 2 cores to memcached would be a waste of resources for low target QPS. Furthermore, we implemented two thresholds instead of one to prevent 'thrashing,' which often occurs with a single threshold system. Using only one threshold could indeed lead to frequent toggling of core allocation, especially if CPU utilization oscillates around that unique threshold. Our dual-threshold strategy thus effectively minimizes excessive switching between cores and reducing the overhead associated with frequent context switches. Regarding how often we decide to reevaluate the core allocation, we chose to check every 0.24 seconds, as our experiments showed that a lower value implies too much overhead because of the frequent context switches and on the other hand, a higher value leads to often breaking the SLO because of a too slow reaction to load changes. Finally, regarding our decision of the precise values of our thresholds, it was based on question 1 of part 4 and the graph with CPU utilization and also based on tests. For t1 we picked the value 10 percents and such a low value makes sens because in the figure where CPU utilization is measured for different loads with 1 core and 2 threads, we see the yellow line (95th percentile latency) crosses the red line (SLO) at an achieved QPS for which the CPU utilization is very low. On the other hand, for t2, we picked 27.5 percents (for the sum of CPU utilization on both memcached's cores) because experimentally, it was low enough to allow for not breaking the SLO and still gives 3 cores to the PARSEC jobs as often as possible, therefore maximising efficiency. It has to be noted that although those thresholds are the ones we used for the data we obtained in the plots below, the best value for those thresholds seem to change a little when deleting and recreating the cluster (ie when changing the actual machines the controller runs on).

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

  Answer:

  As explained previously, memcached alternates between having 1 core and 2 cores. As there are 4 cores on the machine and we want to avoid contention on memcached's cores, the PARSEC jobs can therefore have either 3 or 2 cores associated. We hence have 2 queues, the first one (q2) that is scheduled when 2 cores are available for the jobs, and the second one (q3) scheduled when 3 cores are available. To decide which jobs to put in the queues, our reasoning was that we wanted the 2 queues to become empty preferably at a very close point in time. Indeed, having one queue empty a relatively long time before the other would be problematic since, as explained in the next answer, we decided to give to a job the same number of threads as the number of cores it can generally run onto. As a result, on the one hand, if q2 empties before q3, when memcached uses 2 cores, scheduling q3 would mean running a job with 3 threads on 2 cores, and as seen in question 1 of part 4, having a number of threads superior to the number of cores has a non negligible negative impact on performance. On the other hand, if q3 empties before q2, when memcached uses only 1 core, scheduling q2 would mean running jobs with 2 threads on 3 cores, which would either lead to poor resource utilization in case only 1 job is run or to a number of threads greater than the number of cores in case more than 1 job is scheduled, which either way is not optimal. Since the number of threads given to a job is static, in order to statically chose how to populate the queues (and therefore the number of threads given to each job), we therefore modelled our goal of having queues

emptying around the same time, and as earlier as possible with the following formula:

$$(q2^*, q3^*) = \min_{all\_possible\_allocations} \left\{ \min \left(\frac{\sum times\_of\_jobs\_in\_q2}{average\_time\_memcached\_uses\_2\_cores}, \frac{\sum times\_of\_jobs\_in\_q3}{average\_time\_memcached\_uses\_1\_core} \right) \right\}$$

where $q2^*$ and $q3^*$ represent the optimal job allocations for the queues. To solve this equation, we first measured the running time of jobs in the setting with 2 cores and 2 threads available and in the setting with 3 cores and 3 threads available. We also measured the expected time memcached spent having 1 core versus 2 cores. For this, we simply measured the average ratio of those 2 times across multiple runs and as the experiment is quite long, this ratio was quite similar across the different random seeds. We then created a python script named minimize.py that tests all possible queues allocation and gives us the best one according to the above defined objective. The result was to have canneal, ferret, freqmine and radix in q2 and all the other jobs in q3. One thing to further note is that having q2 and q3 emptied at exactly the same time is generally impossible, so in case one empties before the other, special adaptations are considered in order to maintain high resource utilization. In particular, if q2 is emptied before q3 and memcached uses 2 cores, jobs from q3 will sequentially run on only 2 cores (even if it isn't optimal in terms of number of threads versus number of cores). Same, if q3 empties before q2 which still has more than 1 job unfinished and memcached uses only 1 core, 2 jobs from q2 will be scheduled concurrently, one using 2 cores and the other one using 1 core. We could also have kept in this scenario only 1 job scheduled at a time, running on all 3 cores, but given that it has only 2 threads, it wouldn't have been so beneficial and would make a worse solution in terms of both resource utilization and total execution time. Finally, in case q3 empties before q2 and q2 has only 1 job left, it will use all available cores to maximize resource utilization.

- blackscholes: As explained above, blackscholes was placed in q3 and has therefore 3 cores in general (it can also happen that it uses 2 cores in case q2 empties before blackscholes terminates and memcached uses 2 cores at some point).
- canneal: As explained above, canneal was placed as the first job in q2 and has therefore 2 cores in general (the only case where it could have 3 cores is if q3 empties before canneal finishes and memcached uses 1 core at some point, which is quite improbable considering canneal's position in q2).
- dedup: As explained above, dedup was placed in q3 and has therefore 3 cores in general. (it can also happen that it uses 2 cores in case q2 empties before dedup terminates and memcached uses 2 cores at some point).
- ferret: As explained above, ferret was placed as the second job in q2 and has therefore 2 cores in general. (the only case where it could have 3 cores is if q3 empties before ferret finishes and memcached uses 1 core at some point, which is quite unprobable considering ferret's position in q2).
- freqmine: As explained above, freqmine was placed in q2 and has therefore 2 cores in general. In case q3 empties before freqmine finishes, it is also possible that freqmine runs only on 1 core because in case memcached uses only 1 core and q2 contains radix and freqmine, radix and freqmine will run concurrently on 2 and 1 core respectively. We decided to give in this special case 2 cores to radix (on which its 2 threads can spread nicely) and only 1 to freqmine because part 2 of the project showed us that radix's speedup in time scaled better with the number of threads compared to freqmine.
- radix: As explained in the answer for freqmine, radix is placed in q2 and will generally have 2 cores at its disposal (even if running concurrently with freqmine,

as explained just above). The only case in which radix will use 3 cores is if q3 empties before q2, memcached uses only 1 core at some point and radix is the only reamaining job in q2.

- vips: As explained above, vips was placed in q3 and has therefore 3 cores (it can also happen that it uses 2 cores in case q2 empties before vips terminates and memcached uses 2 cores at some point).

- How many threads do you use for each of the batch job? Why? **Answer:**
  We kept a similar reasoning as part 3 for this matter. Therefore, each job had the same number of threads as the number of cores it could run on. The reason behind this choice is that allocating one thread per core ensures each core is fully utilized, maximizing computational efficiency and throughput. Indeed, each tasks being divided across all available cores it reduces idle time and enhances resource utilization for compute-intensive workloads, compared to having less than 1 thread per core. Moreover, it proved also to be better for the performance, since it avoids context switching or thread switching overhead and potential resource contention that would happen in case there were more than 1 thread per core.
  - blackscholes: It is in q3, having 3 cores to run on in general, so 3 threads.
  - canneal: It is in q2, having 2 cores to run on in general, so 2 threads.
  - dedup: It is in q3, having 3 cores to run on in general, so 3 threads.
  - ferret: It is in q2, having 2 cores to run on in general, so 2 threads.
  - freqmine: It is in q2, having 2 cores to run on in general, so 2 threads.
  - radix: It is in q2, having 2 cores to run on in general, so 2 threads.
  - vips: It is in q3, having 3 cores to run on in general, so 3 threads.

- Which jobs run concurrently / are collocated and on which cores? Why?
  Answer:
  Given our design, either q2 is scheduled or q3. Therefore, no jobs are running concurrently in general, even if there can be interleavings. To be more precise on the set of cores where each job runs, in general, q2 is scheduled on cores 2 and 3, and q3 is scheduled on cores 1,2 and 3, while memcached can be scheduled either on core 0 or on cores 0 and 1. The only case in which jobs run concurrently is when q3 finishes before q2, which has more than 1 job unfinished, and memcached uses only 1 core. In this case, 2 jobs from q2 run concurrently, one on cores 2 and 3 and the other one on core 1, to ensure high resource utilization and lower overall execution time. If this happens, it is very likely to be the 2 last jobs from q2 which would run concurrently. Therefore, to pick which jobs from q2 could run concurrently (and would thereofre be placed at the last positions in q2), we decided to pick the jobs that suffer the less from contention on shared resources like the LLC or the memory bandwidth, and based on our measurements from part 2, we chose freqmine and radix. As explained in the question about cores per job, in this case radix is given 2 cores (on which its 2 threads can spread out nicely) and freqmine only 1 because part 2 showed us that radix scales better with the number of threads than freqmine.
  Having jobs run mostly sequentially as explained above enables to respect our principle that the number of threads for a job should be in general equal to its number of available cores (so 2 cores available means it's good to run 1 job with 2 threads and 3 cores available means it is good to run 1 job with 3 cores). We explored the option to have 3 queues: q1' where jobs would have in general 1 thread and 1 core, q2' where jobs would have in general 2 thread and 2 core and q3' where jobs would have in general 3 thread and 3 core. With this scheduling, q1' would have been able to run

concurrently with q2' when memcached uses only 1 core and q3' was already emptied. However, this strategy appeared to be worse in terms of execution time so we switched to our current design using 2 queues.

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

  Order (to be sorted): canneal (q2),blackscholes (q3), ferret (q2),vips (q3), freqmine(q2), dedup (q3), radix (q2)

  Why:

  The order is actually variable. The only fixed order is inside q2 and q3 respectively, meaning it is for example always the case that canneal runs before ferret. However, the interleaving between the queues isn not fixed but depends on the actual load on memcached and cores available to PARSEC jobs, to have a good matching between threads are cores and good CPU utilization. Regarding the order inside the queues, as the jobs run sequentially there is in general no real reasoning about resource contention between jobs or clever choice we can apply here and the order within queues can therefore be changed without affecting noticeably the overall execution time, which was confirmed by our different tests. The only case in which order within the queue matters is in case q3 empties before q2. Indeed, if this happens, as explained above, 2 jobs from q2 will run concurrently. As if this happens, it is very likely going to be the last 2 jobs of q2, we decided to put there (at the end of q2) the jobs we thought were the best fit within q2 to run concurrently, i.e. freqmine and radix because of their relatively low dependency on shared ressources that we discovered during part 2.

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

  Answer:

  One important difference is that in part 3, there were multiple machines and they had different configurations and were therefore more appropriate to certain jobs (for example the cpu high machine was appropriate to cpu intensive jobs). Here, all the jobs have to be scheduled on the same machine so no such considerations can be made. Moreover, in part 3, concurrency among the PARSEC jobs was much more central since all jobs were running with another job in parallel on the same machine, hence picking the right pair of jobs relatively to interference was crucial. In part 4, the only time we faced this issue was in case q3 ends before q2 and a pair of jobs from q2 has to run concurrently. On the other hand, one similarity to note is that in both experiments, we felt the need to have some kind of queues (or succession of jobs) that optimally need to end at the same time to maximize resource utilization and efficiency.

  However, the most central difference is the fact that the core allocation could be changed in part 4. It opened so much scheduling possibilities and most importantly enabled us to use resources more efficiently, particularly regarding memcached where in part 4 we try to measure the core utilization and allocate the least amount of cores to memcached to match the SLO, while in part 3, the static nature of the scheduler makes this core allocation constant and therefore makes a waste of resource when the load is low. Same, changing the core allocation of PARSEC jobs in part 4 enables to almost never have an idle core, while in part 3 it was almost impossible to avoid. Indeed, having no idle cores in part 3 would mean timing the last scheduled jobs to all end perfectly at the same time, since the static nature of the scheduler meant we couldn't allocate the cores of an exited job to another one already running. In part 4, we are able to dynamically change the core affinity (eg we can adapt, with 2 jobs from q2 that can exceptionally share 3 cores in case q3 is empty and 3 cores are available). Moreover, the ability to pause and unpause jobs, in part 4, also favors a more efficient use of resources since we can schedule the right queue adapted to the current core allocation of memcached and pause the jobs from the other queue, something we couldn't do in part 3 since a job had to run uninterrupted from start to completion.

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

  Answer:
  Our implementation is in the script named `dynamic_experiment.py`. The `dynamic_experiment.py` script's uses the Docker Python SDK and its first task is to fill q2 and q3 with their respective PARSEC jobs whose containers are created, using the `create()` method and detailing the container name, initial core allocation, Docker image, and execution command, which includes the number of threads. Then, memcached's core allocation is first set to 1 core. This is achieved by identifying memcached's PID by scanning active processes and then applying the `taskset` command to assign the appropriate CPU cores. Then, a while loop is entered every 0.24 second during which 2 important steps occur.

  The first one is adapting memcached's core allocation based on its resource need. The decision is based on its CPU core utilization that is monitored using `psutil.cpu_percent`. Based on the CPU usage that is compared to hard coded threshold values, and the current number of cores used, memecached's allocation is therefore adjusted, again using the `taskset` command.

  The second important step occurs with the call of the `reschedule`'s method, with the current core usage of memcached passed as an argument. This method determines which queue to activate on which cores and the number of jobs to run concurrently —typically one, but adjustments are made depending on available cores and the status of job queues (for example, if q3 is empty and 3 cores are available, 2 jobs from q2 run concurrently, one using 2 cores and the other one using 1 core).

  During the rescheduling process, necessary jobs have their CPU cores settings updated using Docker's `update` method and are then unpaused with Docker's `unpause` method. Concurrently, other jobs that shouldn't run are paused using Docker's `pause` method. Additionally, in the main while loop that runs every 0.24 seconds, `dynamic_experiment.py` calls its method `clean_all_queues` to manage the removal of exited containers using Docker's `remove` method and maintains the job queues by removing jobs that have completed.

  The main loop in `dynamic_experiment.py` also continuously checks if all jobs are completed by checking if both queues are empty before leaving 60 seconds for memcached to run alone and ending the experiment.

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

  Answer:
  We think the design choices are already quite detailed in the previous question. Regarding the trade-offs we faced, the major one was between respecting the SLO and achieving the lowest overall execution time. It was mostly reflected in how we chose the values of our 2 thresholds. Indeed, raising their values makes memcached spend a higher proportion of time with only 1 core, therefore potentially breaking more often the SLO, but at the same time, it enables to have more cores for the PARSEC jobs that consequently exit earlier on average. A balance had therefore to be found experimentally to mostly respect the SLO while keeping a low overall execution time. Another trade-off we faced occurred for the decision of the frequency of the while loop in our controller. Indeed, a high frequency meant a good reaction to load variations hence potentially a better respect of the SLO, but it also increased the overhead of rescheduling which involves pausing jobs and changing core allocations.

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.

job name	mean time [s]	std [s]
blackscholes	42.33	0.5
canneal	165.61	4.52
dedup	15.02	0.65
ferret	196.68	5.54
freqmine	244.19	0.74
radix	21.07	0.79
vips	37.72	0.63
total time	716.6	7.87

Answer:

Run Number	SLO violation ratio [%]
Run 1	0.0
Run 2	0.0
Run 3	0.0

Table 1: SLO violation ratio for memcached.

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, 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`).

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

Plots:

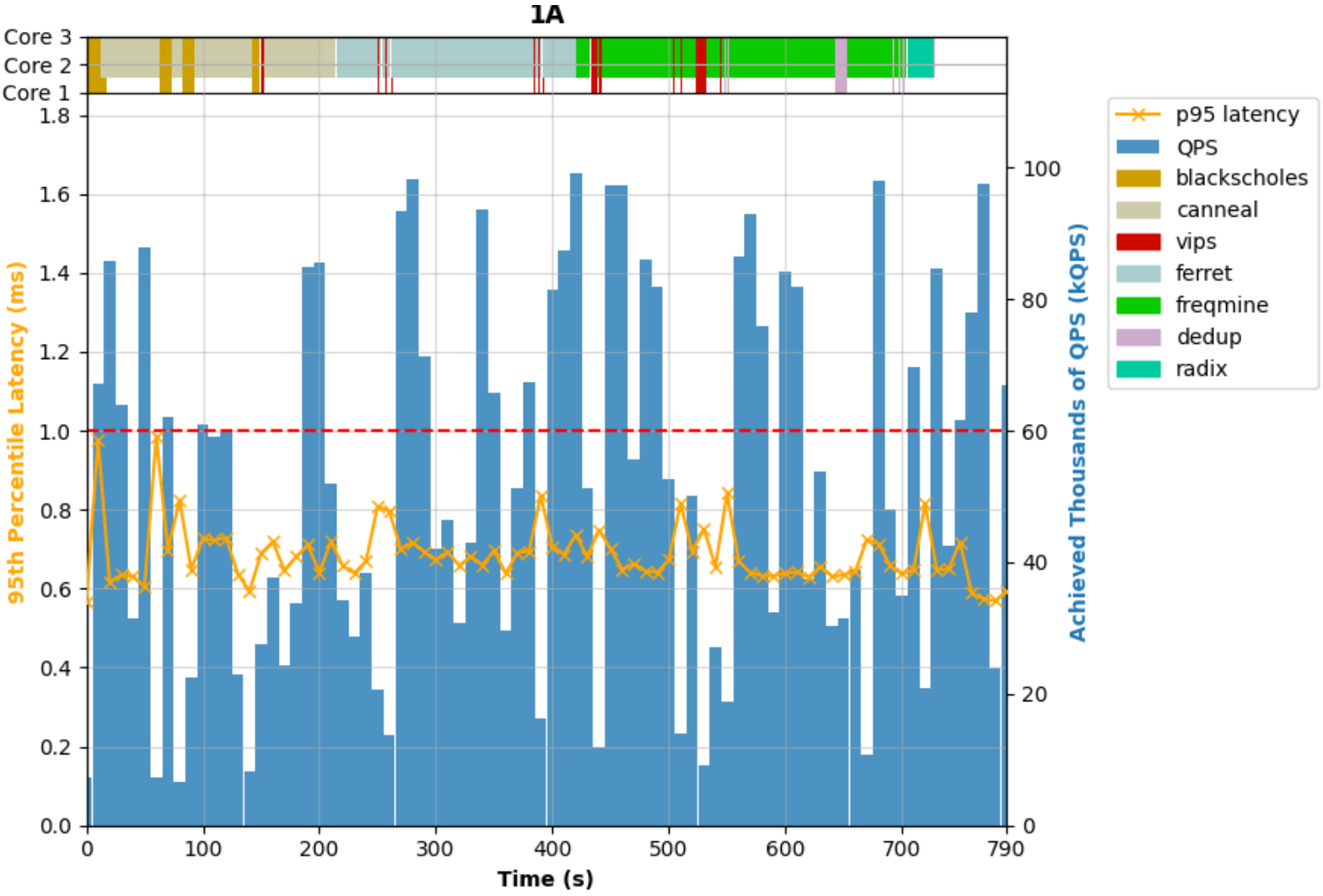
1A
Core 3
Core 2
Core 1
95th Percentile Latency (ms)
Achieved Thousands of QPS (kQPS)
Time (s)
p95 latency
QPS
blackscholes
canneal
vips
ferret
freqmine
dedup
radix

Figure 7: 1A

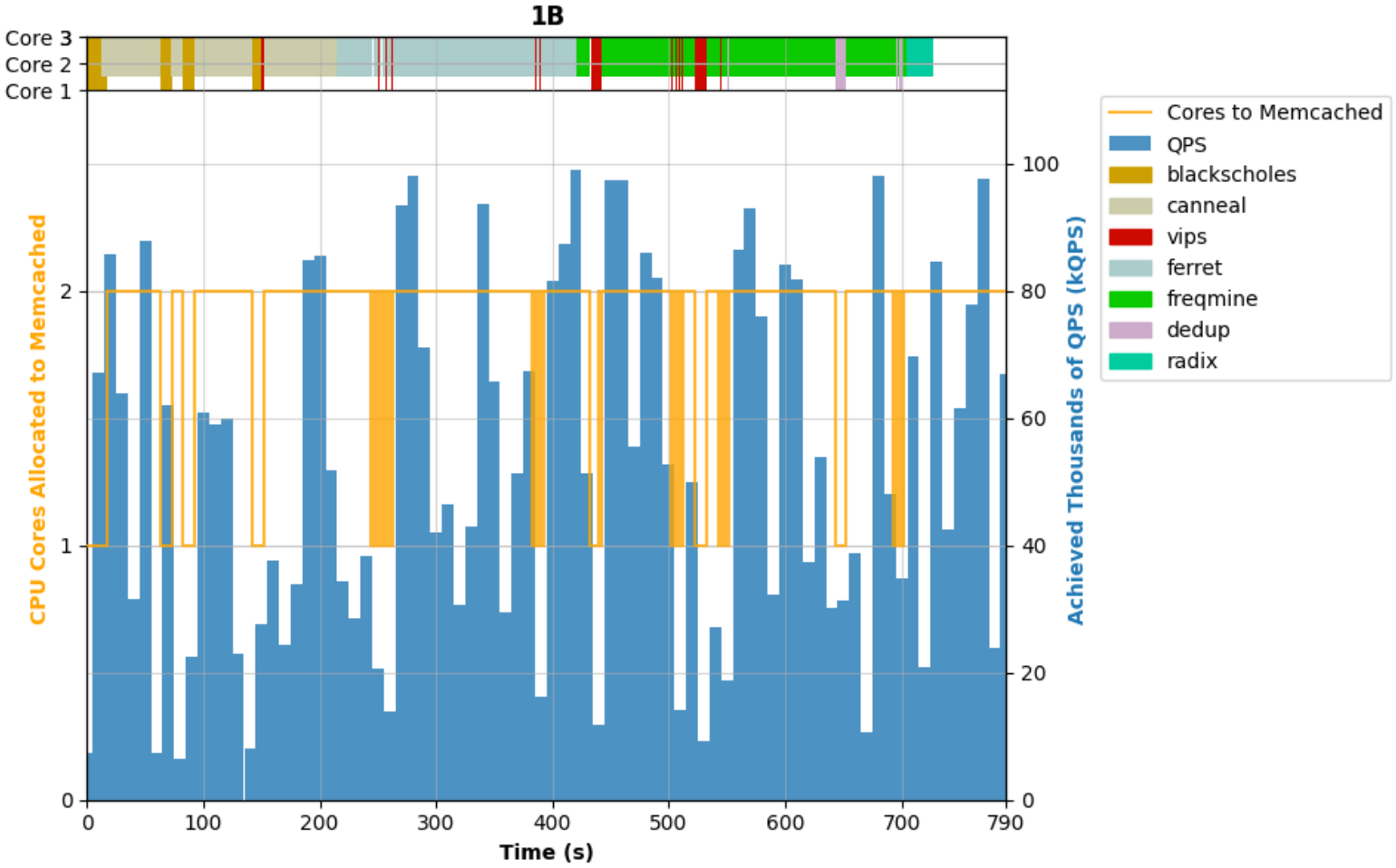
1B
Core 3
Core 2
Core 1
CPU Cores Allocated to Memcached
Achieved Thousands of QPS (kQPS)
Time (s)
0
100
200
300
400
500
600
700
790
0
1
2
0
20
40
60
80
100
Cores to Memcached
QPS
blackscholes
canneal
vips
ferret
freqmine
dedup
radix

Figure 8: 1B

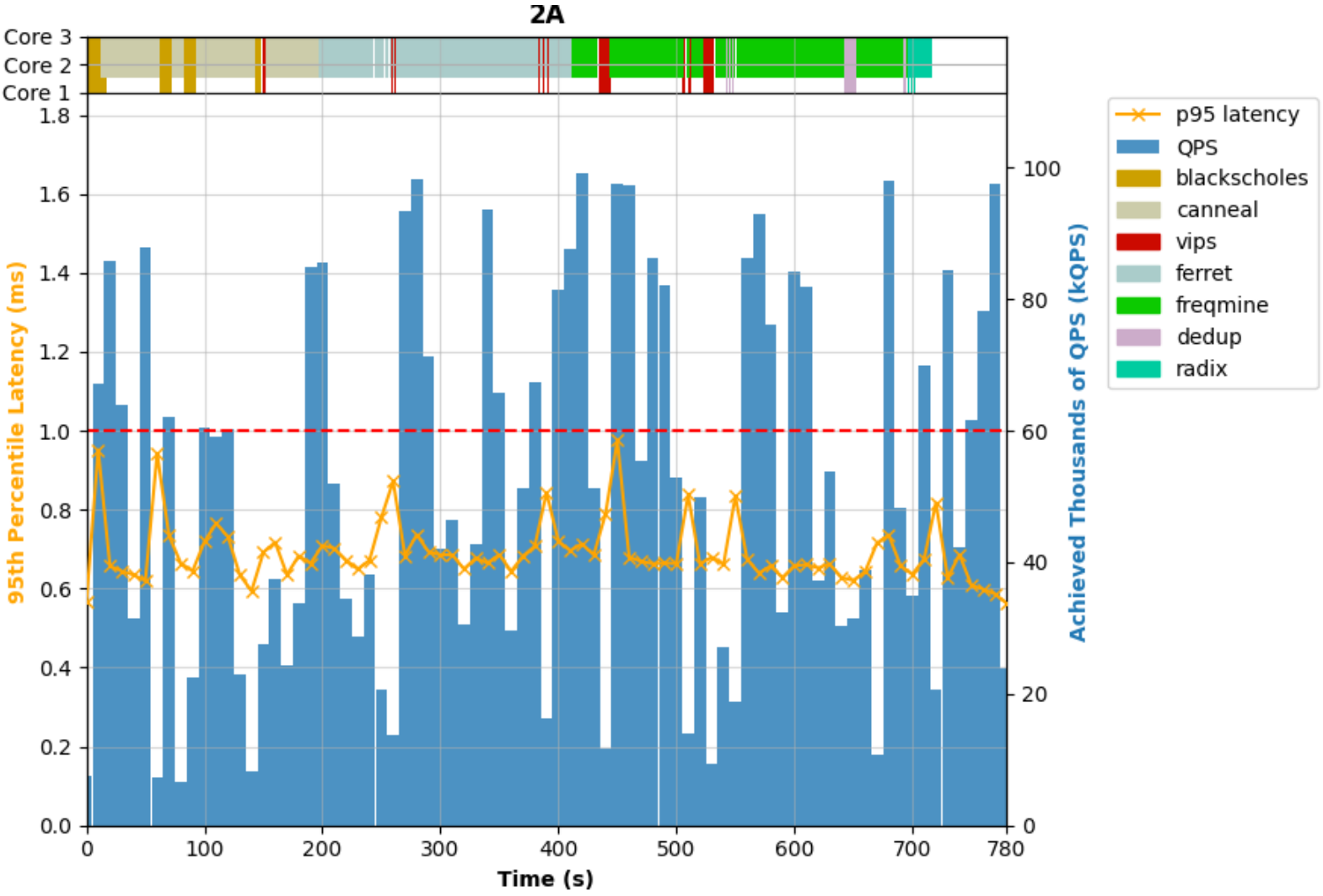
2A
Core 3
Core 2
Core 1
95th Percentile Latency (ms)
Achieved Thousands of QPS (kQPS)
Time (s)
p95 latency
QPS
blackscholes
canneal
vips
ferret
freqmine
dedup
radix

Figure 9: 2A

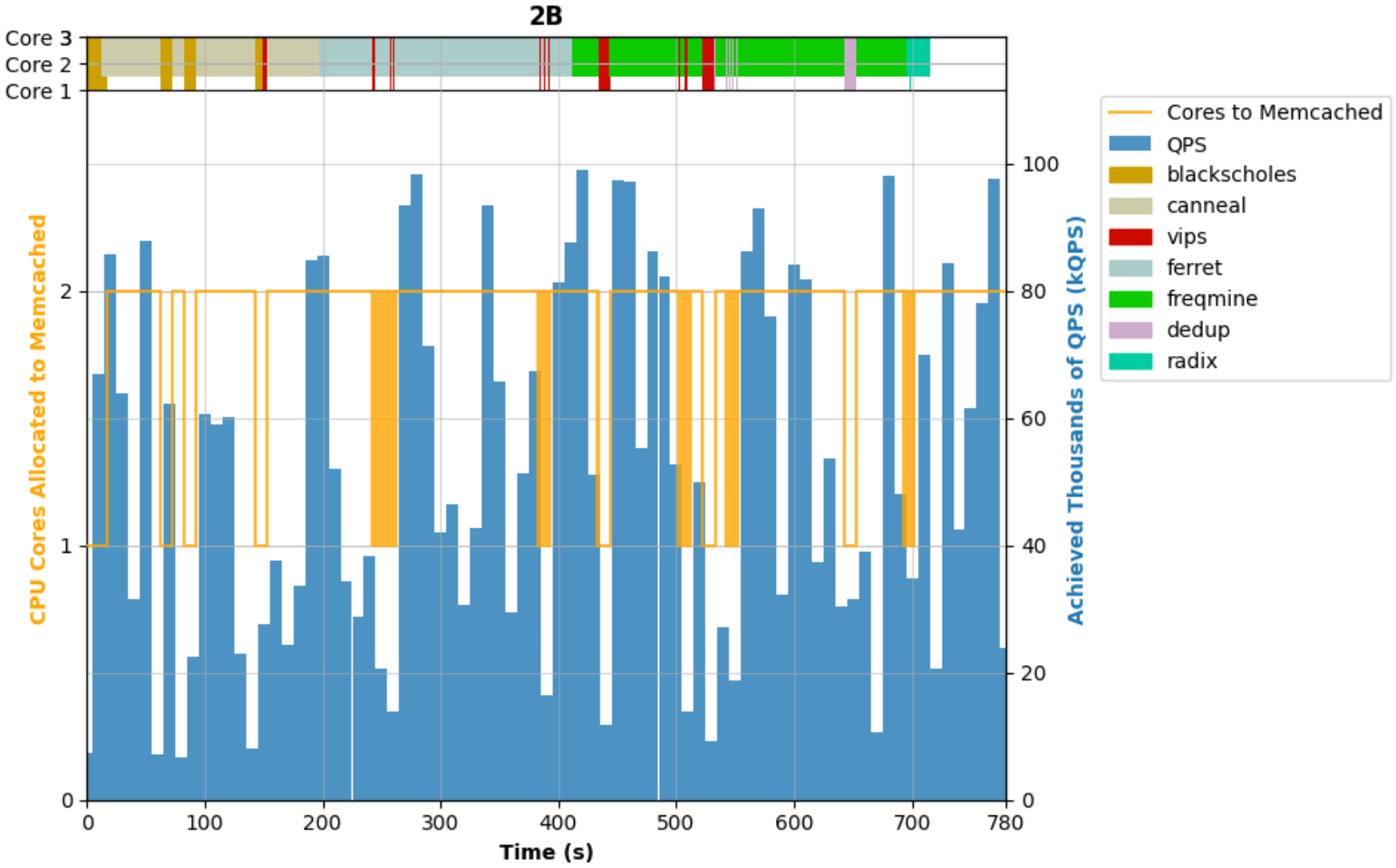
2B
Core 3
Core 2
Core 1
CPU Cores Allocated to Memcached
Achieved Thousands of QPS (kQPS)
Time (s)
0
100
200
300
400
500
600
700
780
0
1
2
0
20
40
60
80
100
Cores to Memcached
QPS
blackscholes
canneal
vips
ferret
freqmine
dedup
radix

Figure 10: 2B

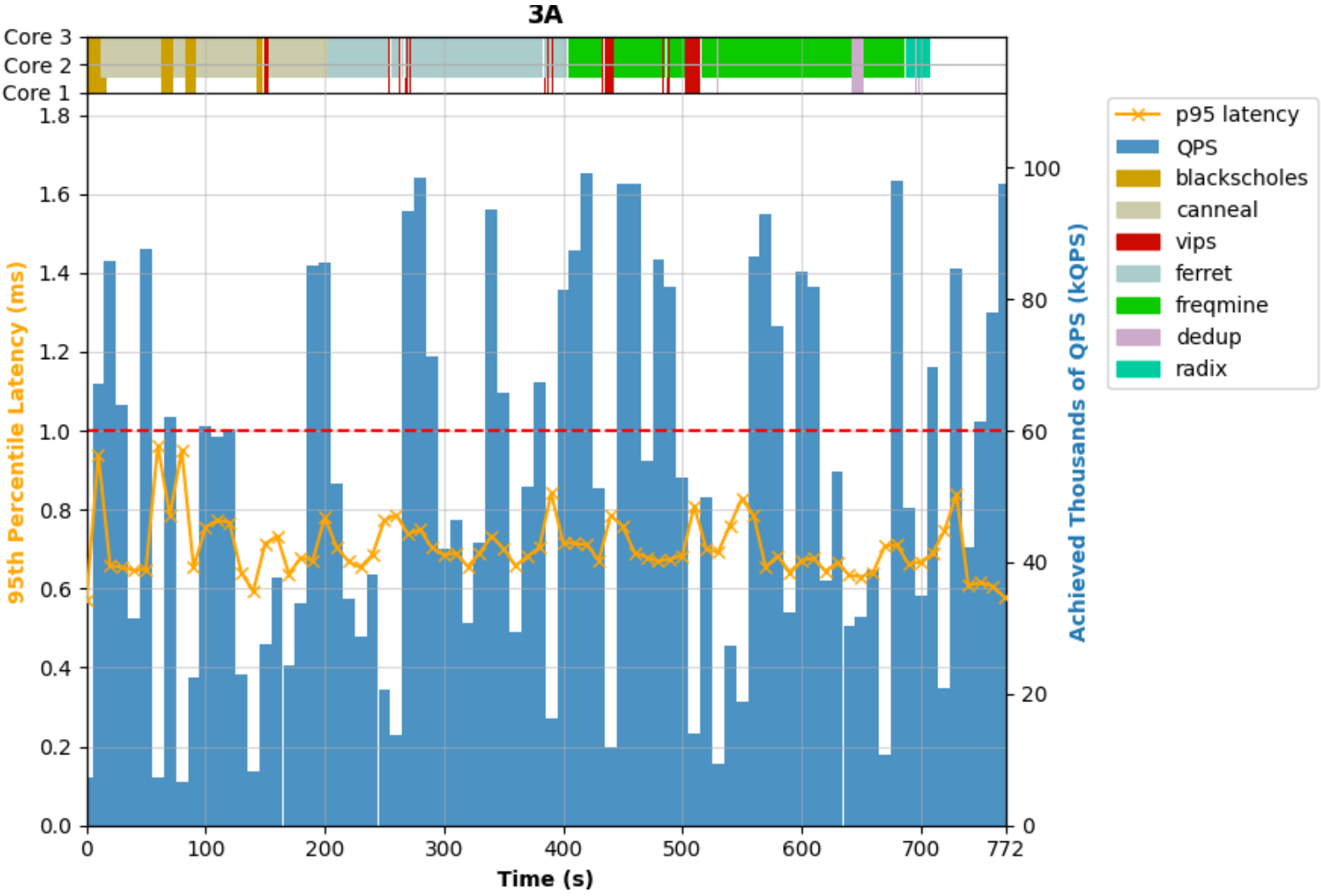
3A
Core 3
Core 2
Core 1
95th Percentile Latency (ms)
Achieved Thousands of QPS (kQPS)
Time (s)
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6
1.8
0
20
40
60
80
100
0
100
200
300
400
500
600
700
772
p95 latency
QPS
blackscholes
canneal
vips
ferret
freqmine
dedup
radix

Figure 11: 3A

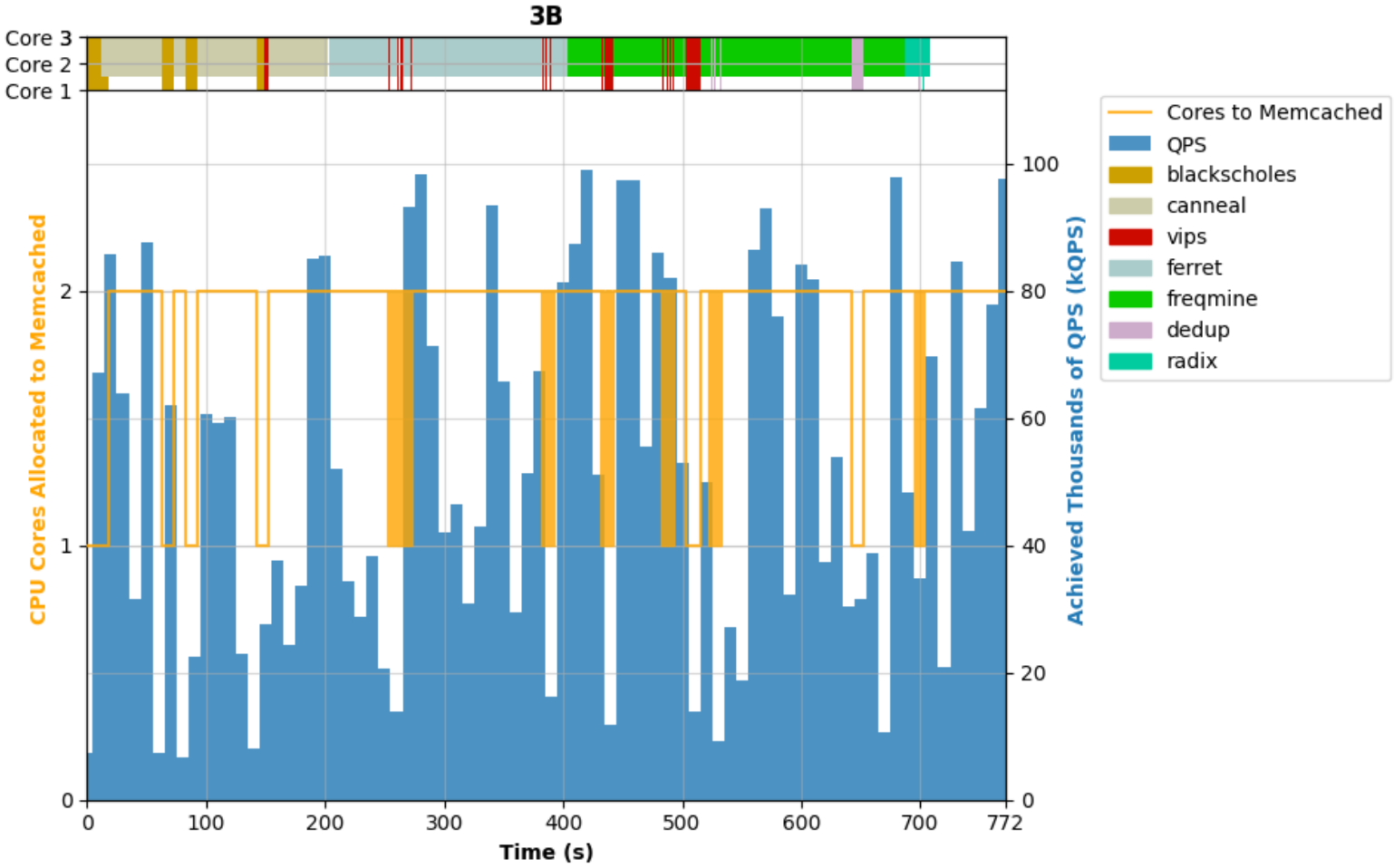

Figure 12: 3B

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:
In case of higher load variability, our policy performs similarly in terms of overall execution time, which remains around 12 minutes without counting the minute with memcached running alone at the end. The SLO violation ratio increases, which is expected as our scheduler is

challenged to be more reactive and give more cores to memcached when there is a high load. Finally, the behavior of the queues is still similar as with a lower load variability in the sense that in both cases, they empty at a relatively close time (within a difference of a minute at most, with q3 emptying just before q2), even though with a smaller interval, the difference increases a little bit.

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:
It is around 1.2 percents, as across our 3 runs the average we obtained was 2 datapoints over 157 that broke the SLO.

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:
With our tests, the smallest interval seems to be 4 seconds, since with it, we obtained across our different runs a SLO violation ratio of 4/186, 4/187 and 3/182, which makes an average SLO violation around 2%. We tried with an interval of 3 seconds, but the 3% margin was not consistently met (for some runs, the ratio went up 5%, even if for others it was around 2.5%).

Run Number	SLO violation ratio [%]
Run 1	2.15
Run 2	2.14
Run 3	1.65

Table 2: SLO violation ratio for memcached with command interval of 4 seconds.

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

Answer:
It is complicated to explain why it is this specific value, although it seems to be the value at which for lower intervals, our scheduler is not reactive enough and ends up gives too few cores to memcached too often. Indeed, the principal features that have an effect on this smallest value are the 3 values that we could qualify as hyperparameters in our scheduler. The first value is the frequency at which the while loop of our scheduler is executed, in which we potentially change memcached's cores and reschedule the jobs. This value influences greatly the reactiveness of our scheduler to the changes of loads. To illustrate this, if we imagine the extreme case where the frequency is extremely low (ie we sleep for a time higher than the interval specified in the command), our scheduler would often not even measure the CPU utilization within a period of a specific load and therefore not adapt the core allocation, hence breaking the SLO for high loads. The other 2 important hyper parameters are the 2 thresholds t1 and t2 described previously and to which the CPU utilization is compared to allocate more or less cores to memcached. Setting these thresholds very low would lead to a higher overall execution time, but since memcached would have more often 2 cores, the SLO would be more respected, even for lower intervals specified in the command. Again, to illustrate, if at the

extreme we set those 2 thresholds to 0, memcached would always have 2 cores and would always meet the SLO, even for very small intervals. An hypothesis for explaining this specific value of 4 seconds interval is that when the load command is launched at the very same time as our controller, given our sleep value of 0.24 seconds in our controller's while loop, we measured across the whole experiment the average time difference between the time when a new target QPS for memecached is set up and the time our while loop is reentered (ie the time we take before readapting the scheduling). For a command interval of 4 seconds, this average time difference was around 0.09 seconds, while for a command interval of 3 seconds, it was around 0.12 seconds. This can be a measure of the reactiveness of our scheduler and might explain why a command interval of 3 seconds breaks the 3% SLO violation margin since the numbers show that our scheduler takes longer to react when the 3 seconds interval is used, and in case of a new high load being set up, more queries will experience a high latency before we readapt memcached's resources, hence leading to a worse ratio.

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	40.2	0.59
canneal	164.11	0.91
dedup	16.69	0.71
ferret	197.78	3.4
freqmine	238.85	1.51
radix	20.8	0.74
vips	38.03	0.51
total time	707.47	8.9

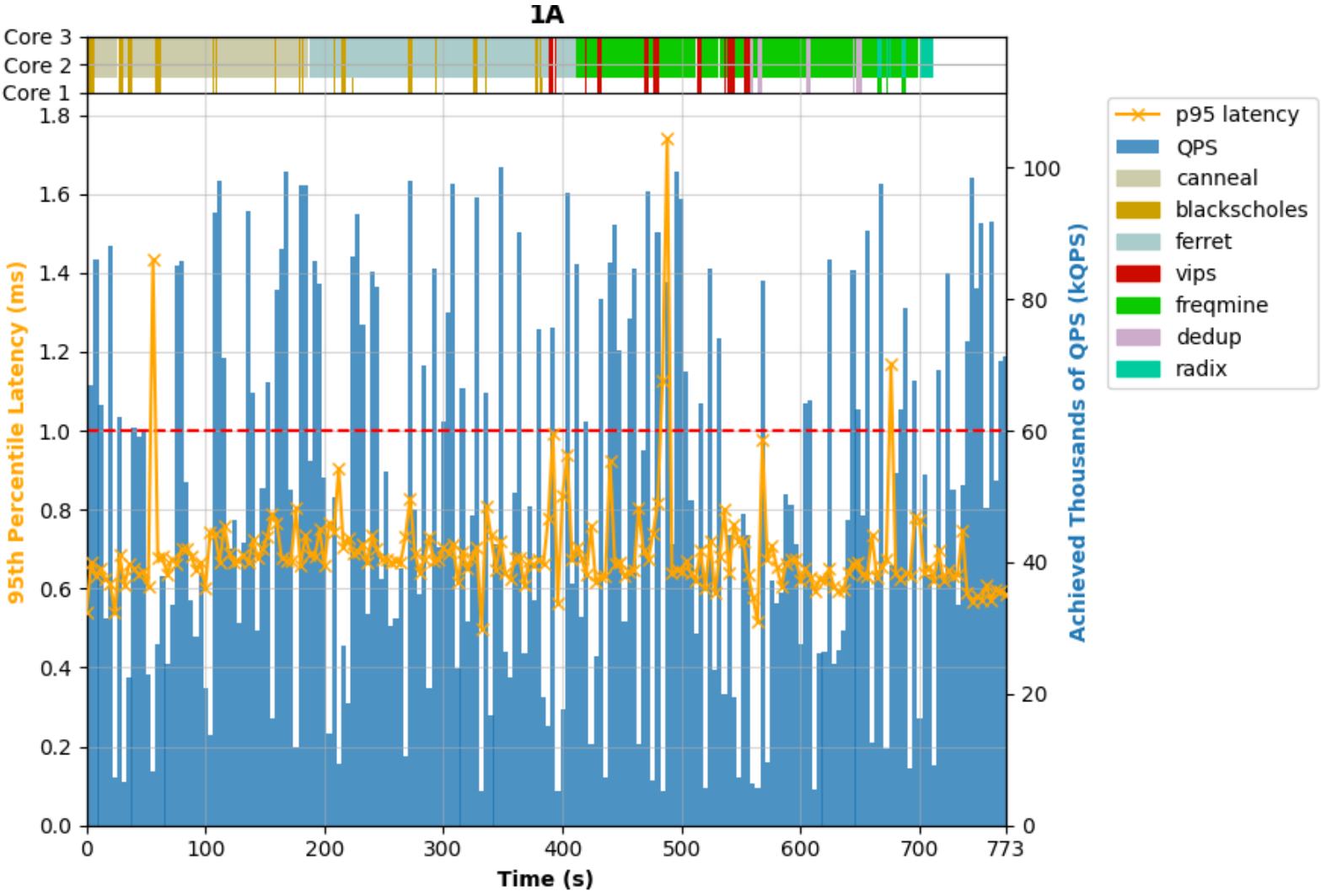
1A
Core 3
Core 2
Core 1
95th Percentile Latency (ms)
Achieved Thousands of QPS (kQPS)
Time (s)
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6
1.8
0
20
40
60
80
100
0
100
200
300
400
500
600
700
773
p95 latency
QPS
canneal
blackscholes
ferret
vips
freqmine
dedup
radix

Figure 13: 1A

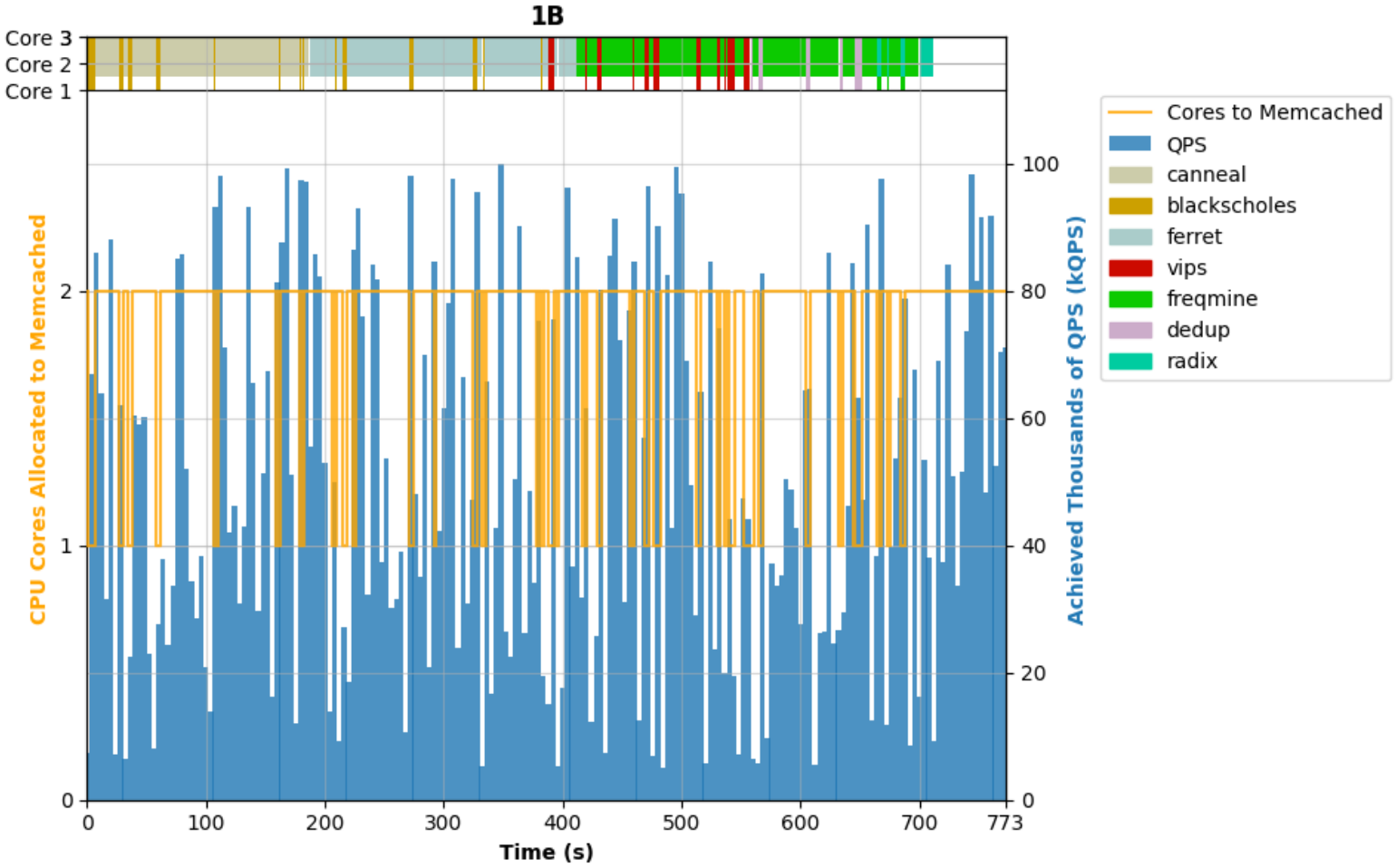
1B
Core 3
Core 2
Core 1
CPU Cores Allocated to Memcached
Achieved Thousands of QPS (kQPS)
Time (s)
0
100
200
300
400
500
600
700
773
0
1
2
0
20
40
60
80
100
Cores to Memcached
QPS
canneal
blackscholes
ferret
vips
freqmine
dedup
radix

Figure 14: 1B

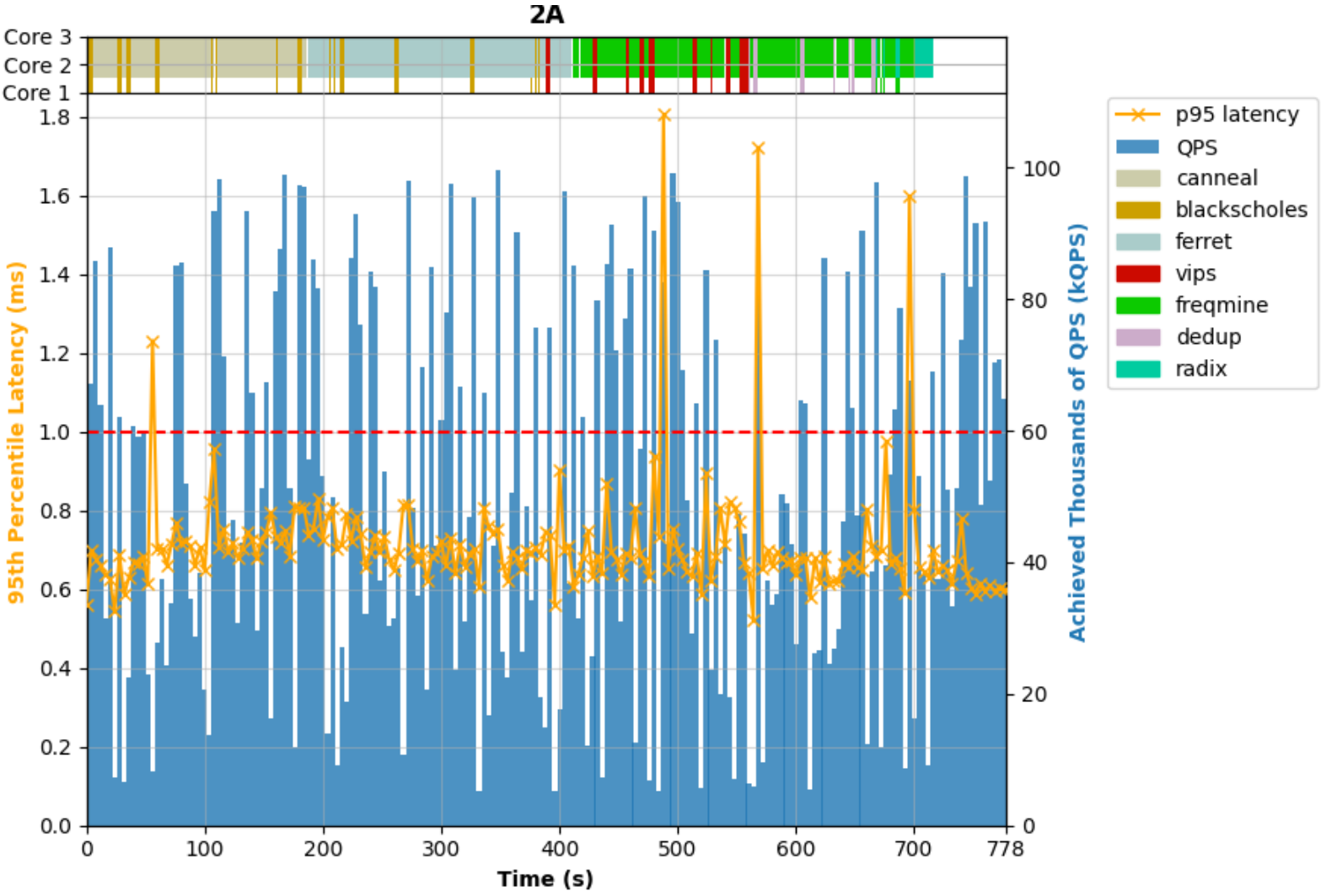
2A
Core 3
Core 2
Core 1
95th Percentile Latency (ms)
Achieved Thousands of QPS (kQPS)
Time (s)
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6
1.8
0
20
40
60
80
100
0
100
200
300
400
500
600
700
778
p95 latency
QPS
canneal
blackscholes
ferret
vips
freqmine
dedup
radix

Figure 15: 2A

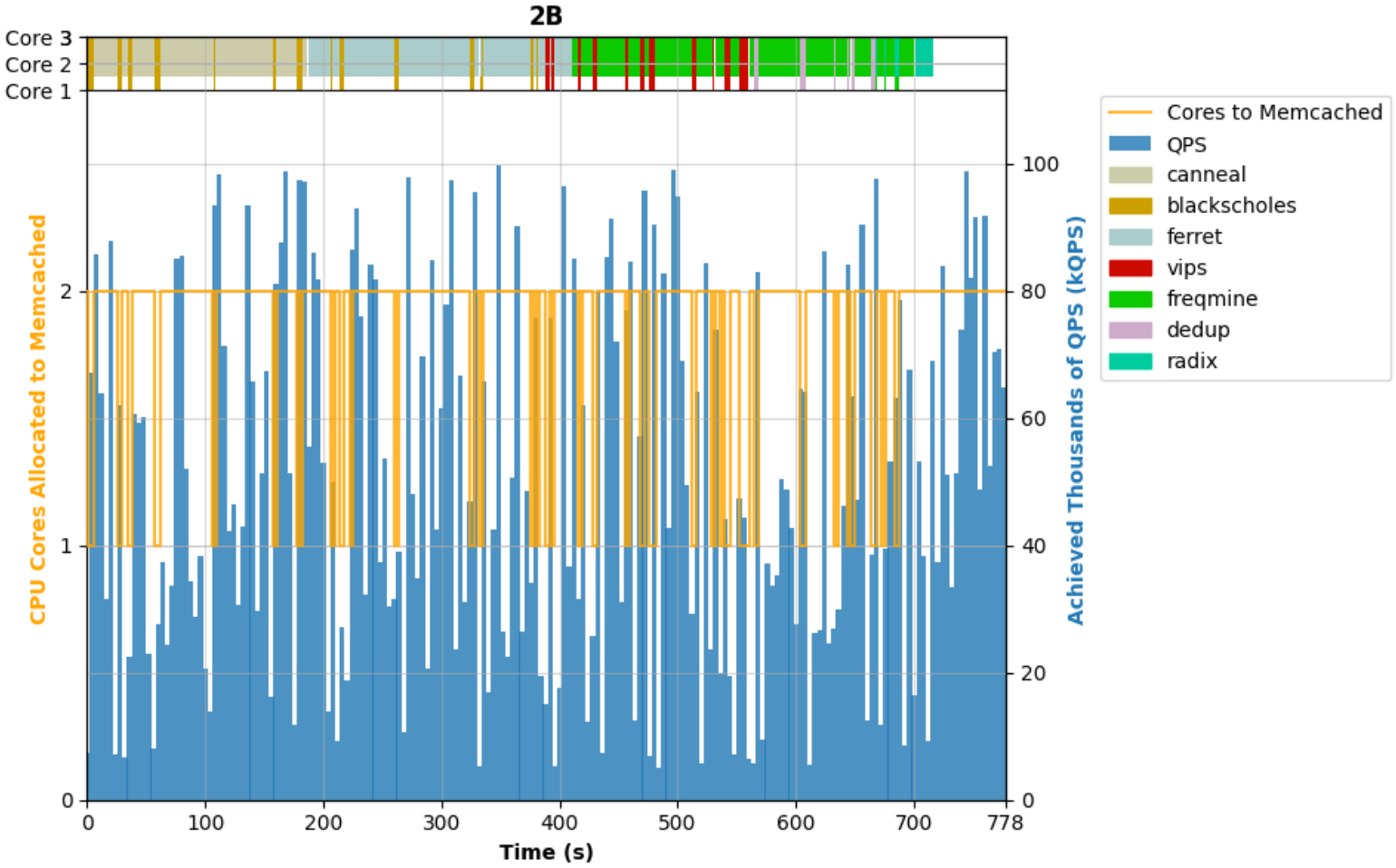
2B
Core 3
Core 2
Core 1
CPU Cores Allocated to Memcached
Achieved Thousands of QPS (kQPS)
Time (s)
0
100
200
300
400
500
600
700
778
0
1
2
0
20
40
60
80
100
Cores to Memcached
QPS
canneal
blackscholes
ferret
vips
freqmine
dedup
radix

Figure 16: 2B

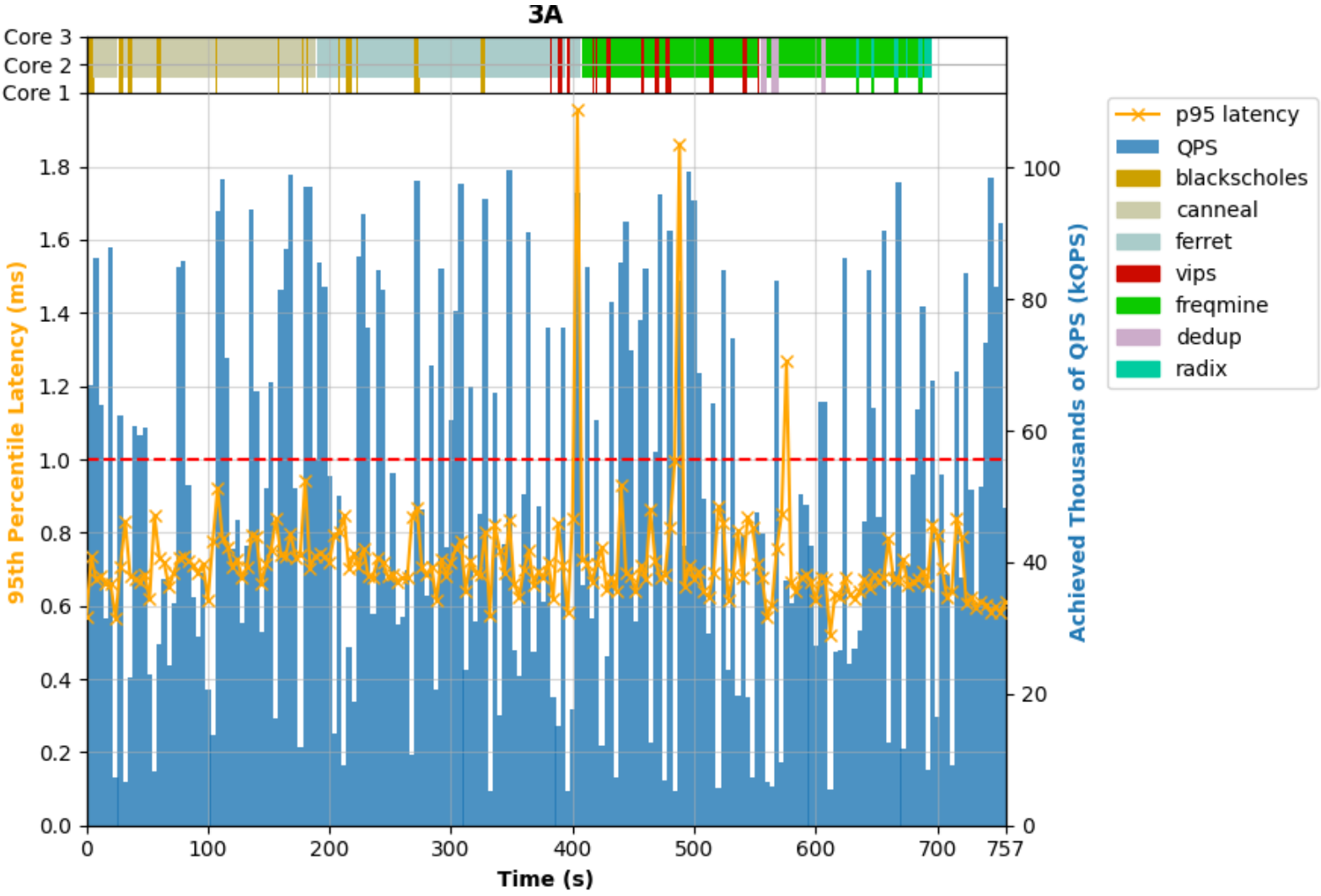
3A
Core 3
Core 2
Core 1
95th Percentile Latency (ms)
Achieved Thousands of QPS (kQPS)
Time (s)
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6
1.8
0
20
40
60
80
100
0
100
200
300
400
500
600
700
757
p95 latency
QPS
blackscholes
canneal
ferret
vips
freqmine
dedup
radix

Figure 17: 3A

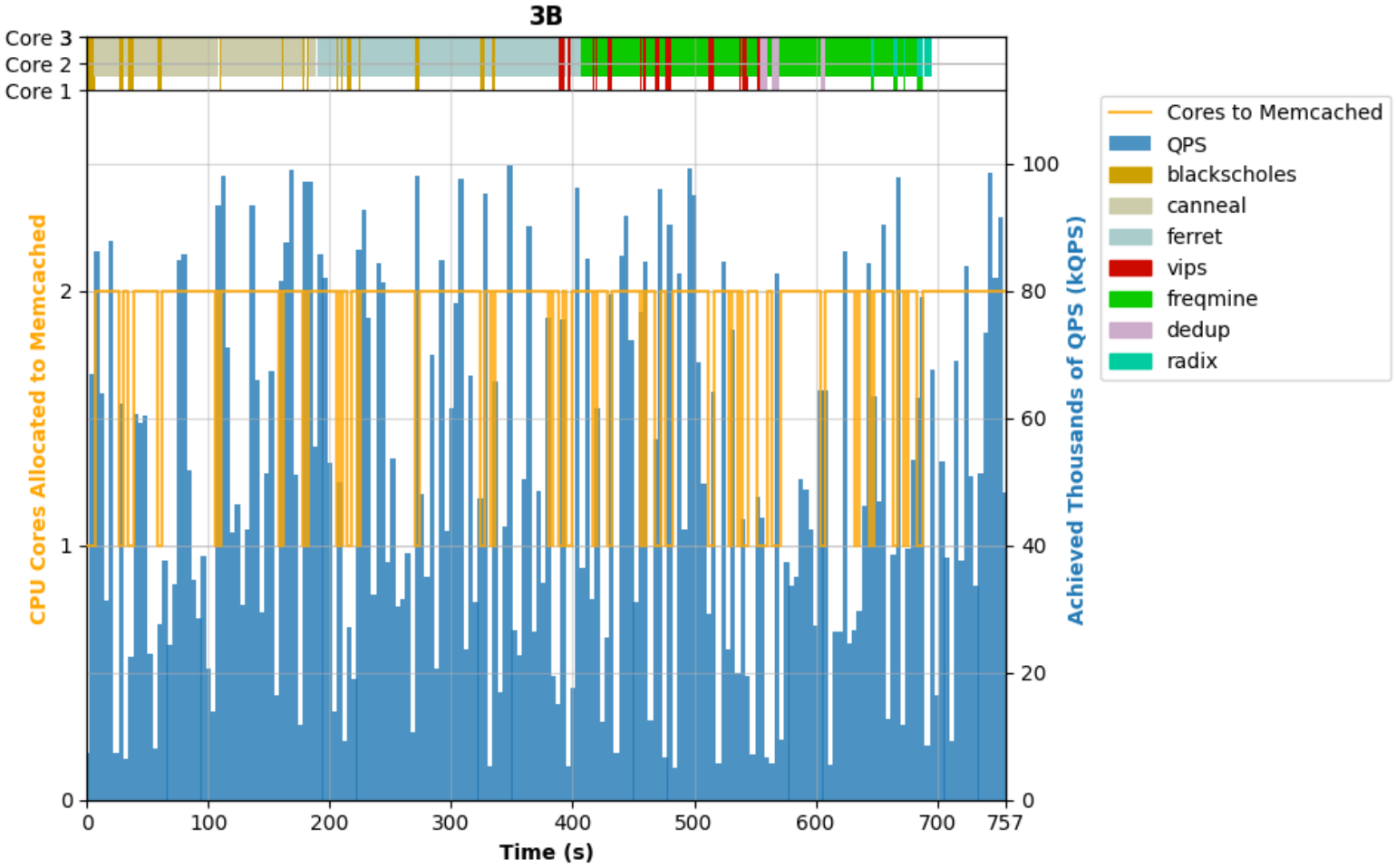
3B
Core 3
Core 2
Core 1
CPU Cores Allocated to Memcached
Achieved Thousands of QPS (kQPS)
Time (s)
0
100
200
300
400
500
600
700
757
0
1
2
0
20
40
60
80
100
Cores to Memcached
QPS
blackscholes
canneal
ferret
vips
freqmine
dedup
radix

Figure 18: 3B