Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Cloud Computing Architecture

Semester project report

Group 048
Luca Tagliavini - 23-955-461
Christopher Zanoli - 23-942-394
Rudy Peterson - 23-941-701

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

Part 3 [34 points]

1. **[17 points]** With your scheduling policy, run the entire workflow **3 separate times**. For each run, measure the execution time of each batch job, as well as the latency outputs of memcached, running with a steady client load of 30K QPS. For each batch application, compute the mean and standard deviation of the execution time [1] across three runs. Also, compute the mean and standard deviation of the total time to complete all jobs - the makespan of all jobs. Fill in the table below. Finally, compute the SLO violation ratio for memcached for the three runs; the number of data points with 95th percentile latency > 1ms, as a fraction of the total number of data points. The SLO violation ratio should be calculated during the time from when the first batch-job-container starts running to when the last batch-job-container stops running.

Answer:

job name	mean time [s]	std [s]
blackscholes	114.33	1.15
canneal	139.33	0.58
dedup	10.33	0.58
ferret	94.33	0.58
freqmine	140.67	0.58
radix	10.00	0.00
vips	29.33	0.58
total time	153.67	0.58

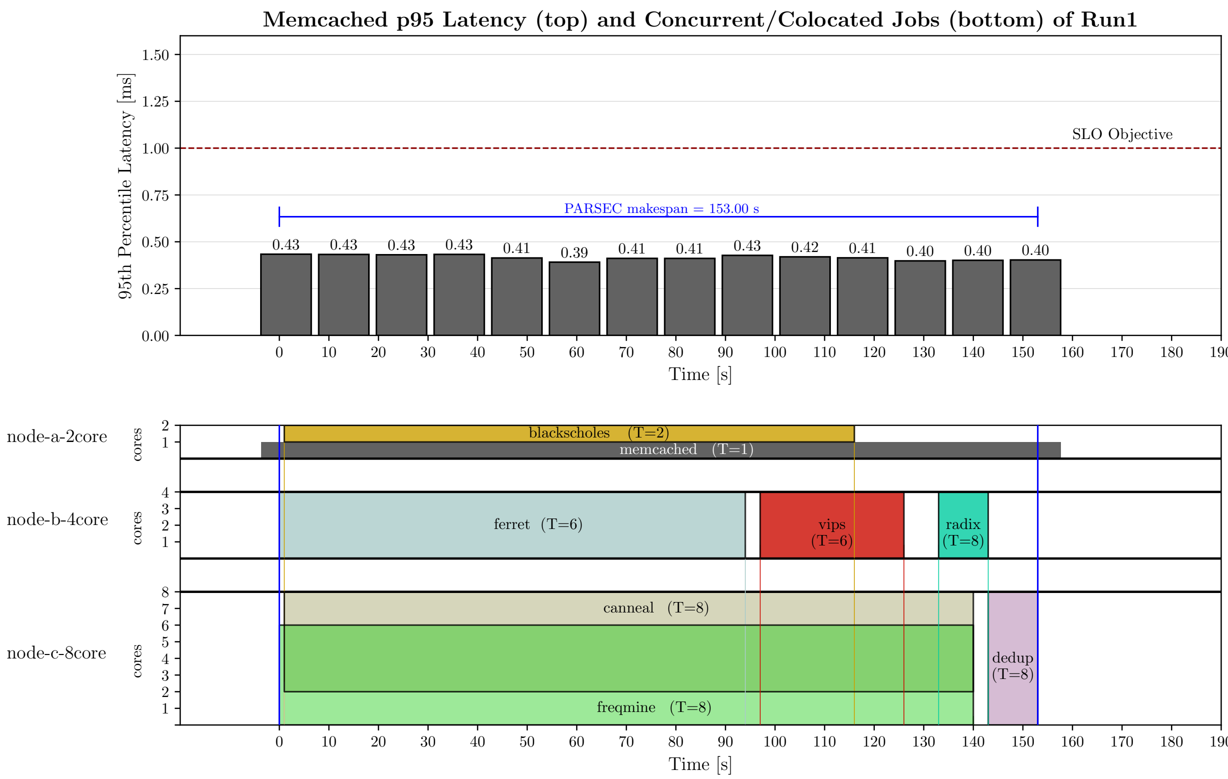
The following table shows the SLO violation ratios for each of the three runs:

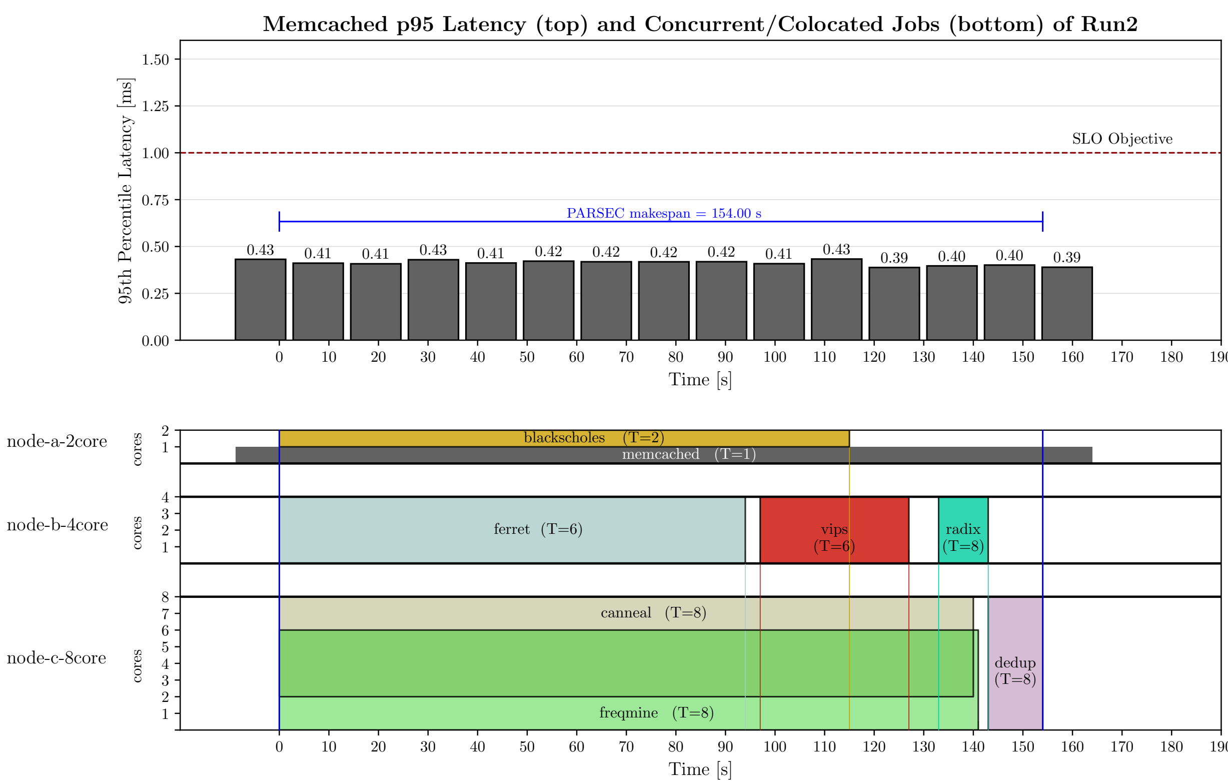
Run 1	Run 2	Run 3
$\frac{0}{17}$ (0%)	$\frac{0}{18}$ (0%)	$\frac{0}{17}$ (0%)

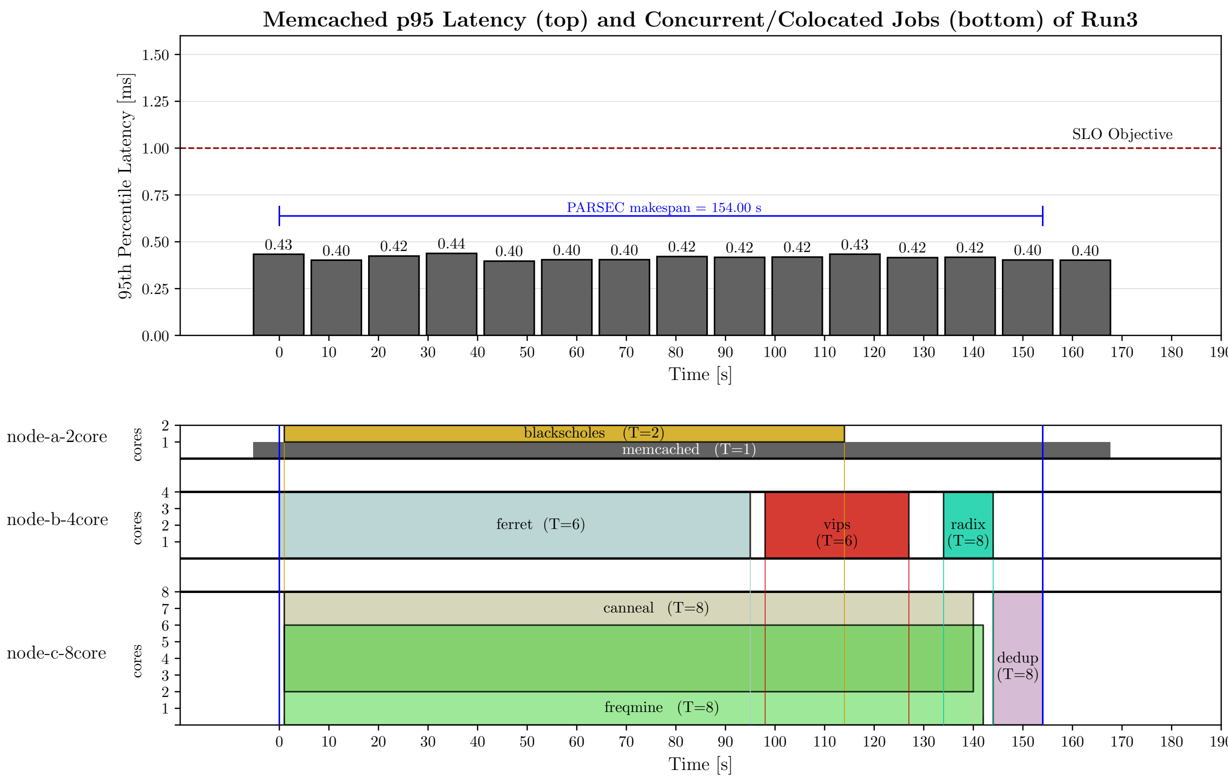
Create 3 bar plots (one for each run) of memcached p95 latency (y-axis) over time (x-axis), with annotations showing when each batch job started and ended, also indicating the machine each of them is running on. Using the augmented version of mcperf, you get two additional columns in the output: `ts_start`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


Memcached p95 Latency (top) and Concurrent/Colocated Jobs (bottom) of Run1
95th Percentile Latency [ms]
SLO Objective
PARSEC makespan = 153.00 s
0.43 0.43 0.43 0.43 0.41 0.39 0.41 0.41 0.43 0.42 0.41 0.40 0.40 0.40
Time [s]
node-a-2core
blackscholes (T=2)
memcached (T=1)
node-b-4core
ferret (T=6)
vips (T=6)
radix (T=8)
node-c-8core
canneal (T=8)
dedup (T=8)
freqmine (T=8)
cores


Memcached p95 Latency (top) and Concurrent/Colocated Jobs (bottom) of Run2
95th Percentile Latency [ms]
SLO Objective
PARSEC makespan = 154.00 s
0.43 0.41 0.41 0.43 0.41 0.42 0.42 0.42 0.42 0.41 0.43 0.39 0.40 0.40 0.39
Time [s]
node-a-2core
blackscholes (T=2)
memcached (T=1)
node-b-4core
ferret (T=6)
vips (T=6)
radix (T=8)
node-c-8core
canneal (T=8)
dedup (T=8)
freqmine (T=8)
cores



Note: in the above plots we separated job allocation from the p95 latency bar-plot to improve readability (the x-axis between the 2 sub-plots is aligned). In the lower section of the plot, representing concurrent and colocated jobs, `T` denotes the number of threads allocated to each job. The y-axis of the bottom subplot specifies the individual cores assigned to each job and is annotated with the machine each job is running on. We ensured proper synchronization between the batch jobs and the `mcperf` data; the x-axis aligns with the start time of the first batch job's container. Meanwhile, `memcached` and `mcperf` are running from before the first batch job begins and continues running after the last batch job concludes. In the bottom subplot, vertical lines with the same color as the respective batch job identify the start and end of the respective batch job. As illustrated by the partially overlapped horizontal bars, `freqmine` and `canneal` share 4 cores on the `node-c-8core` machine. We also included the blue line identifying the obtained makespan (minimization objective). Note also that, as requested, the width of each bar in the bar plot is calculated as the difference of the respective `ts_end` and `ts_start` (i.e., the `ts_span`). Thus, each of them has a variable width, which is ≈ 10 ms.

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

   - Which node does memcached run on? Why?

     Answer: `memcached` runs on the `node-a-2core` node, which has the fewest CPU cores and the least memory among the available machine types. We selected this node for two primary reasons: first, `memcached` exhibited low resource requirements; second, the two-core configuration of the node limits job parallelization, which helps achieve a shorter makespan. As a result, this machine is ideal for tasks like `memcached` and parsec-blackscholes, which do not benefit significantly from multithreading and have minimal resource demands.

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

  Answer:

  - blackscholes: It runs on node-a-2core. This setup is ideal because we observed that blackscholes requires minimal RAM usage. With 2 GB of RAM available, this machine represents a suitable choice, leaving sufficient memory for memcached. The 2-core configuration aligns with the batch job speedup pattern we observed in Part 2b of the project (we referred to both the measurements files and the speedup plot): we noticed that we would not obtain a significant gain in the job runtime by assigning more than 2 threads, so one core is allocated to blackscholes while the other is reserved for memcached. For more information, see the final question in Part 3.2.
  - canneal: It runs on node-c-8core. Based on the measurements from Part 2b of the project, this is one of the longest-running jobs among those provided, although it doesn't scale very well. To minimize the overall makespan, we aimed to allocate as many threads (and consequently cores) as possible. Our design strategy involved using 8 logical threads distributed across 6 cores to optimize the balance between runtime and resource utilization. Consequently, canneal could only be placed on node-c-8core, as it is the only machine capable of providing the required number of cores. Observations from the speedup plot in Part 2b of the project indicated that the scaling of canneal is not linear, leading us to conclude that the allocated cores would not be fully utilized. Therefore, we concurrently ran freqmine on 4 of the 6 cores to make efficient use of the available resources.
  - dedup: It runs on node-c-8core. Contrary to the measurements gathered in Part 2b of the project, for dedup we empirically discovered that using 8 cores slightly reduced the overall runtime for this job. As a result, we placed it on the node-c-8core machine to minimize its runtime. Interference with other jobs was not considered in this case, as it would be the only job running on the machine at the time of execution.
  - ferret: It runs on node-b-4core. Two primary requirements guided the placement of this job on node-b-4core: the necessity to avoid interferences and the need to provide 4 cores. These requirements were informed by the job's medium runtime and its effective scaling with an increasing number of threads. To strike an optimal balance between runtime and resource usage, we allocated 4 cores and 6 threads. Given that ferret is highly susceptible to interference, as shown by the data in Part 2a of the project, we opted to avoid colocating it with other jobs, such as canneal and freqmine on node-c-8core.
  - freqmine: It runs on node-c-8core. Along with canneal, this is one of the longest-running jobs in the set. Therefore, parallelizing its execution was crucial for reducing the makespan. As shown in the speedup plot from Part 2b of the project, freqmine scales effectively with the number of threads, so we allocated 8 threads and 6 cores. This is why we selected node-c-8core for this task. Although some cores will be shared with canneal, the interference will be minimal since canneal will not fully utilize all the allocated resources (please refer to the previous point for more detailed reasoning).

  - radix: It runs on `node-b-4core`. This job is one of the quickest in the set. Despite this, it scales well with an increasing number of threads (from Part 2b of the project), so we allocated 8 threads. As a result, a machine with at least 4 cores was required. Since `node-b-4core` only hosts `ferret` and `vips`, we chose this machine because it would have the fastest job queue clearance. As the job is not run concurrently with other jobs, interference was not considered.
  - vips: It runs on `node-b-4core`. The rationale for this placement is similar to that for `radix`. Due to our other scheduling decisions, the job queue on the `node-b-4core` machine will clear faster than that on the `node-c-8core`, thereby minimizing runtime. Consequently, we assigned 4 cores to this job and opted not to run it concurrently with others. Given that the workload benefits significantly from multithreading (from Part 2b of the project), we allocated 8 threads to it. This approach allowed us to disregard any potential interferences that might otherwise impact the job's performance.
- Which jobs run concurrently / are colocated? Why?
  - On `node-b-4core`, `ferret`, `vips` and `radix` are sequentially colocated. This combination was chosen because we aimed to allocate 4 cores to each job. However, based on the runtime data from the measurements in Part 2, these jobs do not require concurrent execution. Additionally, our decision to avoid concurrency (and thus prefer sequentiality) was influenced by the significant interference observed between `ferret` and `vips`.
  - On `node-c-8core`, `canneal` and `freqmine` are concurrently colocated. We chose to run these jobs concurrently due to their prolonged durations and significant benefits from parallelization. Given that minimizing the overall makespan of the batch jobs was imperative, this strategy proved essential. Empirical experimentation led us to allocate 4 shared cores between the two jobs, striking a balance between parallelization for faster runtime and the mitigation of interference-related slowdowns.
    Furthermore, we colocated `dedup` on the same node, scheduling it to run sequentially after `canneal` and `freqmine` completed. Running `dedup` concurrently with `canneal` and `freqmine` was not an option, since the resources are already heavily utilized, and `dedup` itself would benefit from having 4 to 8 dedicated cores/threads. Therefore, we arranged for it to start sequentially, utilizing all 8 cores once the preceding jobs concluded. Although performance data from Part 2b of the project indicated that `dedup` experiences diminishing returns from additional cores, this issue did not arise under this specific scenario. Ultimately, our decision to schedule `dedup` sequentially on this node was also influenced by the observation that the combined runtime of `canneal` and `freqmine` was the shortest compared to the makespan of jobs running on other nodes. Therefore, placing `dedup` on a different node would have likely resulted in an increased overall makespan or led to interferences with other jobs.
  - On the `node-a-2core` node, `blackscholes` is concurrently colocated with `memcached`. Our measurements show that `memcached` requires no more than one core to manage its workload effectively. Consequently, we allocated the remaining core to `blackscholes`, with 2 threads. A further justification for the colocation with memcached comes from measurements obtained in Part 2b of the project, where `blackscholes` demonstrated an adequately short runtime with 2 threads. Running `blackscholes` on this separate core proved to be a good design choice also because, in this way, we could avoid any core-related interferences (CPU, L1i, L1d, L2) with

the concurrent running `memcached`. This arrangement is advantageous since, as determined in Part 2a of the project, `blackscholes` causes only minor interference on `membw` and `llc`, which do not significantly affect `memcached`'s p95 latency, as established in Part 1 of the project.

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

  Order:

  (a) `node-a-2core`: blackscholes
  (b) `node-b-4core`: ferret, vips, radix
  (c) `node-c-4core`: freqmine and canneal (in parallel), dedup

  Why: `blackscholes`, `ferret`, `freqmine` and `canneal` are started together as soon as the scheduler begins (thus the specific order provided before is irrelevant, and may change between runs due to uncontrollable delays). This decision was driven by several factors:

  – `blackscholes` is among the longest-running jobs, and should be started as soon as possible
  – Both `canneal` and `ferret` should terminate as soon as possible since it is crucial to free up resources for the subsequent job, i.e. `dedup`.
  – The early launch of `ferret` ensures its timely completion, facilitating the progression of sequentially queued jobs.

  The other jobs are allocated to either 4-core or 8-core nodes, immediately following the completion of their predecessors. This scheduling approach is designed to minimize the makespan effectively. It is noteworthy that the scheduling order of `vips` and `radix` is not crucial; interchanging these tasks doesn't affect the final performance.

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

  Answer:

  – blackscholes: In our empirical assessments, we employed 2 threads on a single core, as this configuration demonstrated a reduction in runtime compared to using only one thread per core.
  – canneal: We employed 8 threads for running `canneal`, despite its relatively poor scaling characteristics, because using this number of threads was necessary to minimize runtime as much as possible. Given the inefficient scaling, the resources allocated were not fully utilized. To optimize resource usage, we concurrently scheduled `freqmine` with it, also with 8 threads, thereby capitalizing on the available spare resources. This strategy ensured a more efficient allocation of computing power under the constraints of `canneal`'s scaling limitations.
  – dedup: We employed 8 threads. Although in our initial measurements of Part 2b of the project this appeared to slow down the execution, this didn't happen in this most recent measurement. Since the machine has all its 8 cores available when `dedup` is scheduled to run, it was reasonable to make full use of all the available resources, assigning 8 threads and 8 cores to it.
  – ferret: We employed 6 threads. Again, the machine only has 4 cores, but we found, through empirical tests, that over-allocating the number of threads actually lowers the makespan.

– freqmine: We employed 8 threads for the job, which was allocated 6 cores, 4 of which were shared with `canneal`. Empirical testing demonstrated that using a slightly higher number of threads than cores available improved runtime. Although `freqmine` benefits from an increased thread count, its scaling is sub-linear. Consequently, the additional threads that could not be fully utilized due to sub-linear scaling were effectively leveraged by running `canneal` concurrently on the shared cores. This approach enabled more efficient use of resources between the two jobs.

– radix: We employed 8 threads for the scheduling of `radix`. The only options for radix are 2, 4 and 8, as it requires a power of two for the number of threads[2]. We decided to use 8 threads as we noticed a minor improvement in performance through our empirical assessment. This observation aligns with the fact that `radix` scales very well with the number of threads, as demonstrated in Part 2b of the project.

– vips: We employed 6 threads. Although the process is allocated only 4 cores, using 6 threads empirically proved to improve the runtime.

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

  Answer:

  – We modified `get_time.py` to also print the start time, which is required for the plot 3.1. In particular it turned out to be crucial for computing the relative start times of the batch jobs with respect to each other, for a correct timeline representation of the jobs in the bottom subplot. It was also crucial for the correct data synchronization with the `mcperf` files.

  – We modified the `yaml` files in the `parsec-benchmarks` folder to:

    (a) Run each job using `taskset`, so that we could specify the affinity for the PARSEC process.

    (b) Specify a `nodeSelector` policy, to force a job to run on a specfic node.

    (c) Specify resources constraints, such as `requests` and `limits`. The former is used to force the scheduling of the batch jobs to follow an hardcoded sequence (along with the scheduling order) and to prevent or allow jobs to be colocated in parallel.

  Consequently, we used the following kubernetes features: `nodeSelector`[3], `resources.requests`[4], `resources.limits`[5].

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

  Answer: In general, we noticed that using between 2 and 1.5 times as many threads as cores usually decreases the runtime. We suppose this is due to some bursty nature of the workloads, which do not fully utilize the given resources at all times. By over-allocating the number of threads and even running jobs on the same cores that are used by other

[2] This was empirically assessed by trying all thread sizes between 1 and 8. Sizes that are not a power of 2, such as 3 and 6, seem to never terminate.

[3] `https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/#nodeselector`.

[4] `https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/#requests-and-limits`.

[5] `https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/#requests-and-limits`.

jobs, we allow the Linux Kernel to do a better and more granular scheduling, as well as optimize the usage of the system's resources even further. Further, this reasoning is backed by the fact that no job managed to achieve a linear scaling in Part 2b of the project, thus meaning that they are generally not able to use all allocated resources. This doesn't mean that giving more threads is a bad idea: on the contrary, using many threads on jobs such as `canneal`, which scales rather poorly, was key for achieving a low makespan. While the scaling is not ideal, the runtime deduction is still significant, and this sub-optimal resource usage implies that cores can be shared by multiple jobs (as long as interferences remain minor) to improve resource utilization and increase parallelism, ultimately reducing the overall makespan.

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:

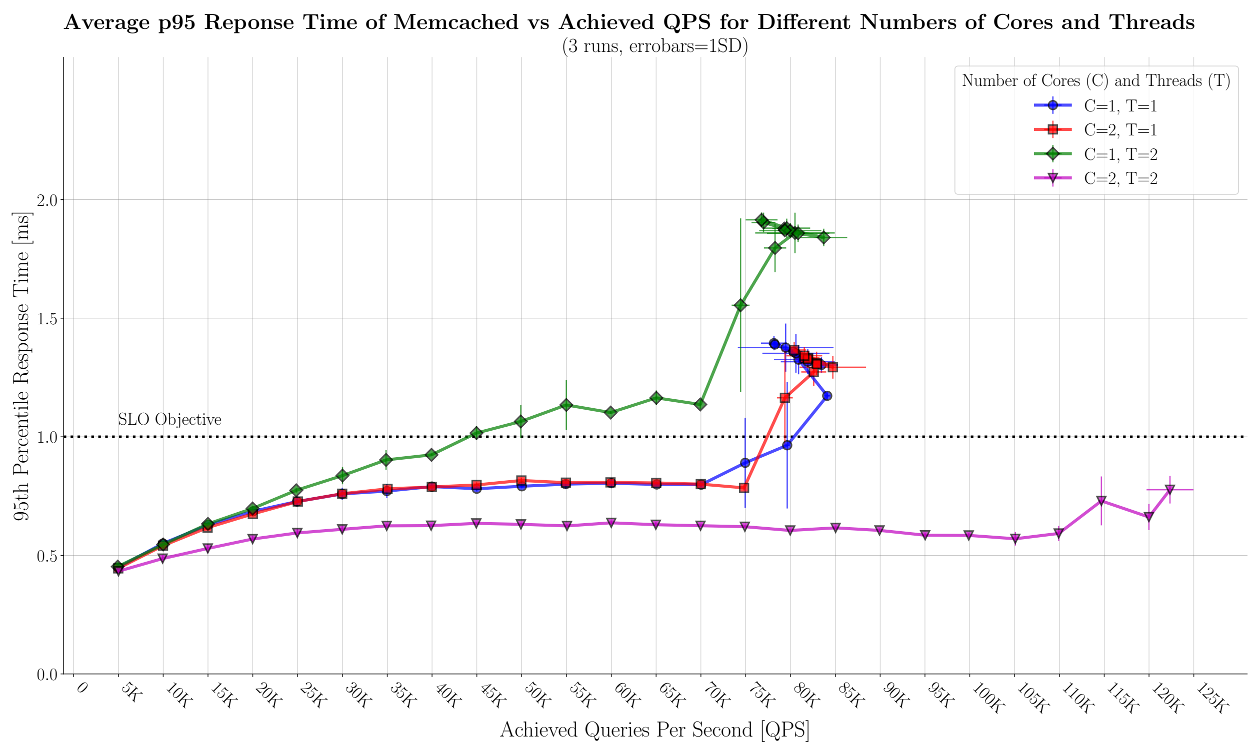


Figure 1

What do you conclude from the results in your plot? Summarize in 2-3 brief sentences how memcached performance varies with the number of threads and cores.

Summary:

The [C=2, T=2] configuration uniquely meets the SLO and achieves the target QPS for memcached.
Other configurations fall short, particularly [C=1, T=2], which breaches the SLO at 45K QPS and saturates at 85K QPS, likely due to resource competition and overhead when threads outnumber cores.
Both [C=1, T=1] and [C=2, T=1] breach the SLO between 75K and 85K QPS and saturate at 85K QPS, suggesting the necessity of multiple threads to leverage multiple cores, as each thread operates on a single core at a time.

b) [2 points] To support the highest load in the trace (125K QPS) without violating the 1ms latency SLO, how many memcached threads (T) and CPU cores (C) will you need?

Answer:
As described in the previous answer, to support the highest load in the trace, a configuration of C=2 and T=2 is necessary, as it is the only configuration capable of achieving 125K QPS without breaching the SLO.

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: The answer is T=2. As described in answer a), configurations with T=1 saturate between 75K and 85K QPS, failing to reach the 125K QPS target. With T=2, however, it is theoretically possible to maintain the 1ms 95th percentile latency SLO across a range from 5K to 125K QPS. Specifically, the configuration C=1, T=2 could be utilized effectively from 5K to $\approx$ 40K QPS. Beyond this QPS, transitioning to the C=2, T=2 configuration sustains performance up to the target 125K QPS.

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:

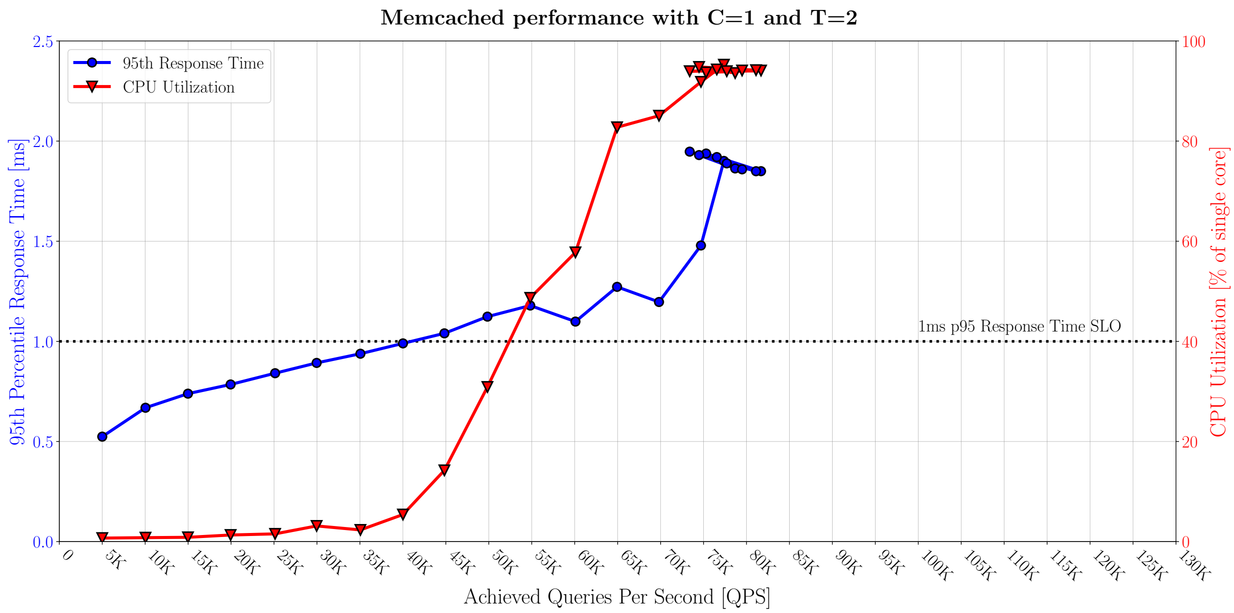


Figure 2: Memcached p95 Response Time and CPU usage of C=1 ($core_0$) and T=1

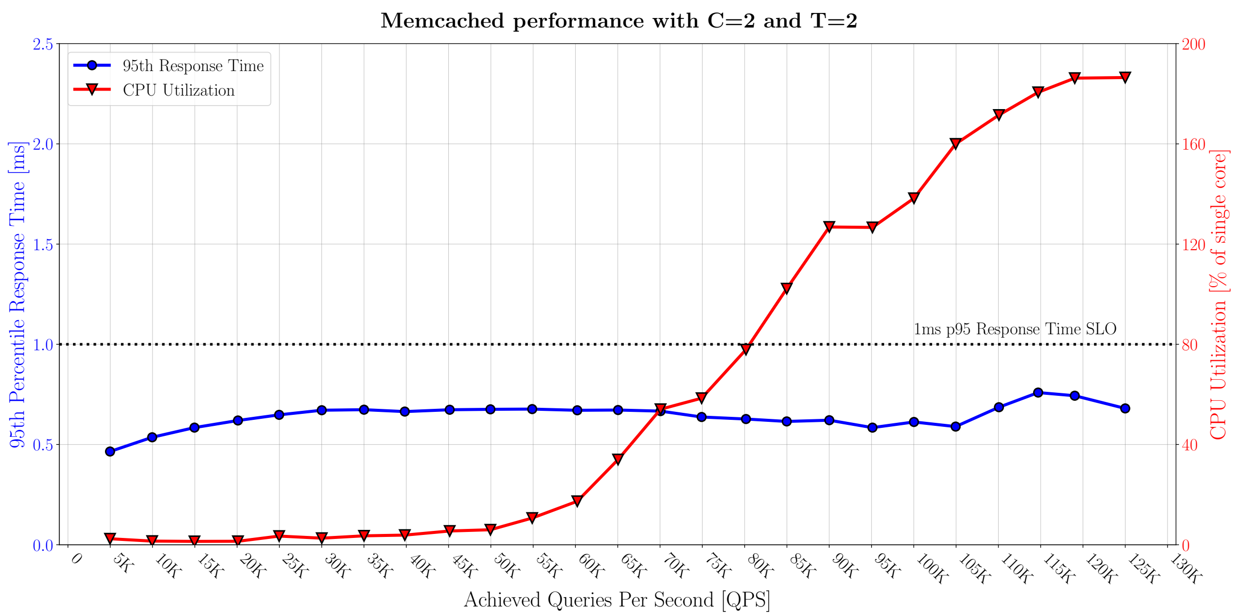


Figure 3: Memcached p95 Response Time and CPU usage of C=2 ($core_0$ + $core_1$) and T=1

Note: The creation of both graphs required a synchronization step between the file containing `mcperf` data and the file containing CPU utilization. Specifically, for each row of the `mcperf` data, the corresponding timestamps from the CPU utilization readings in the interval [`t_start`, `t_end`] were extracted. To associate a single CPU utilization value with the corresponding p95 response time, we calculated the average CPU utilization over that interval. This approach was crucial as CPU utilization values varied within each interval of ≈ 5 s. For

CPU utilization sampling we employed `psutil`[6] and the sampling rate is 100 ms [7].
In order to ensure reliability and reproducibility, the results presented in both charts are derived from the average of three runs, however, in compliance with the specific requirements outlined in the handout, error bars have been omitted.

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.
w 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:
  The policy involves two `AugmentedQueues`[8]: the *high-priority* queue, managing the two cores without memcached, and the *low-priority* queue, managing one core with memcached. These queues track batch jobs to be executed, jobs currently running, and their core assignments. All jobs are executed sequentially. The *high-priority* queue manages two cores, running different jobs in parallel. The *low-priority* queue runs jobs only when requests are below 29K QPS and CPU usage of `core0` + `core1` is below 45%, with `memcached` using one core. If either condition is unmet, jobs are paused and `memcached` gets both cores. Unlike the *low-priority* queue, the *high-priority* queue never pauses jobs and can take jobs from the *low-priority* queue if a core is free. Whenever possible, it can run a single job on both cores. For more details, refer to the last point of Question 4.2. For a visualization, see Figure 4.

[6] `https://pypi.org/project/psutil/`
[7] Based on documentation recommendation `https://psutil.readthedocs.io/en/latest/#psutil.cpu_percent`
[8] For a definition see the last question of this section.

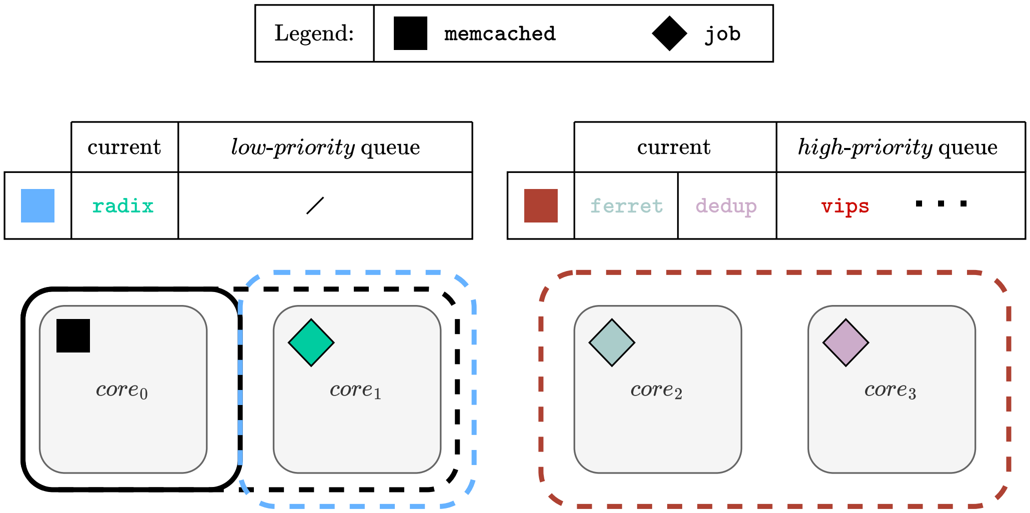


Figure 4: This diagram depicts the initial state of the system after the first three jobs have been scheduled. The *low-priority* queue is empty, while its associated running list contains `radix`. The *high-priority* queue has many jobs waiting, but it also has 2 running jobs: `ferret` and `dedup`. Each job is allocated to a core. This placing is symbolized by the diamond shape of the respective color. `memecached` is currently running on only one core, but the alternative two-core configuration is drawn with a black dashed line. Dashed lines represent the cores on which jobs from a given queue or `memecached` can run on. If `memcached` was using two cores, `radix` would be placed back in the *low-priority* queue's running section, in a paused state.

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

  Answer:

  The scheduler allocates by default two cores to `memcached`, which are required to sustain the load across the whole QPS range. Then, it eagerly tries to reduce the number of allocated cores from two to one, to allow jobs in the *low-priority* queue to execute. This happens when the criteria for low overall system load is met, that is when both:

  (a) The combined CPU usage of $core_0$ and $core_1$, which are the cores where `memcached` is running, is below 45%.

  (b) The number of incoming reads over the last second (namely QPS), is below the threshold of 29K. While in Part 4.1 we determined that `memcached` in the [C=1,T=1] configuration could handle up to 40K requests without breaking the SLO. This number tourned out to be lower in Part 4.2 and onwards, because of two reasons:

    – Some jobs cause interferences, even when they're placed on other cores (i.e, those that interfere on the `llc` or `membw`). This can cause a minor breakage of the SLO, as we've seen in Part 1, so we need to be more conservative.

    – As `memcached` is a networking-heavy application, part of its CPU usage also involves kernel threads that handle the network operations. These threads do not obey `affinity` properties, and thus they can be placed on all cores. Therefore, when we load any core, we are inherently removing resources from `memcached`.

  Finally, to aid system stability, we avoid tasksetting `memcached` to a single core when all jobs in the *low-priority* queue are done.

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

  Answer:

  Statically, the number of cores allocated to each job is determined by the queue it is assigned to, as hard-coded into the scheduler. Jobs placed in the *high-priority* queue will be run on one core by default, while jobs in the *low-priority* queue run on a single core shared with `memcached`. These jobs are not executed concurrently; instead, we use pausing and tasksetting of memcached to optimize system performance. The only exception occurs when there is a single job in the *high-priority* queue; in this case, it is allocated both cores managed by the queue. Additionally, jobs from the *low-priority* queue can be promoted to the *high-priority* queue when the latter is empty. Therefore, at runtime, the number of cores is not fixed. However, in our measurements, this reallocation was never necessary due to our job placement. The static core assignments are outlined below, with indications of when a job may be promoted to two cores.

  - blackscholes: we assigned to it one core because more than one core was not deemed necessary for `blacksholes`, which has a relatively short runtime. Placing it in the *low-priority* queue was not an option though, as the makespan is not short enough. Although this job is in the *high-priority* queue, it has never been promoted to two cores in our experience, as it finishes quite early.
  - canneal: we assign one core to this job. Based on our experience, it completes in approximately 3 minutes, making it unsuitable for the *low-priority* queue. Compared to other jobs in the high-priority queue, it requires relatively little time, so allocating two cores is not necessary. Therefore, it is placed in a queue position where it is unlikely to ever receive both cores.
  - dedup: we assign one core to this job due to its very short makespan. Although it could have been placed in the *low-priority* queue, it would almost always end up being moved to the *high-priority* queue, so we placed it there. Additionally, its position serves to delay the execution of `freqmine` long enough for `ferret` to finish, preventing the severe interferences that occur when these two jobs run simultaneously. To conclude, `dedup` could theoretically also be executed with 2 cores, but it completes so quickly and early that this never happened in our experiments.
  - ferret: we assign one core to this job and place it in the *high-priority* queue because it doesn't take long to complete, but it is not short enough to be placed in the *low-priority* queue. It is positioned first in the queue to ensure it finishes quickly, allowing `freqmine` to start without causing interferences.
  - freqmine: we assign one or two cores to this job, which is placed last in the *high-priority* queue. In our experiments, it has consistently run with both one and two cores. This placement avoids interferences, as previously stated, while the number of cores is chosen to maximize speedup. Since this is the longest-running job, utilizing multiple cores helps reduce the overall makespan.
  - radix: we assign one core to this job, placing it in the *low-priority* queue where it always only receive one core. Although our policy allows it to be moved to the *high-priority* queue and potentially use two cores, this scenario is extremely rare and has not occurred in our experiments. In fact, `radix` always terminates quickly enough to clear the *low-priority* queue before the *high-priority* queue finishes. Further, `radix` manages to terminate this quick while using very little resources, therefore, we allocated only 60% of the CPU time to it, in order to further avoid SLO breaches.

- vips: We assign one core to this job, placing it early in the high-priority queue, where it consistently uses only the default one core and has never utilized both cores in our experience. This decision is based on its short runtime and its ability to run concurrently with `ferret`. The placing of this job is strategic in ensuring sufficient separation between `ferret` and `freqmine`, minimizing interferences.

Furthermore, we emphasize that the ordering of the queue and the number of assigned cores are closely related. The queue order was determined not only by the number of cores assigned to each job but also by their potential interferences. A key decision was placing `ferret` and `freqmine` at opposite ends of the queue. This arrangement prevents them from running simultaneously, as they significantly interfere with each other, ensuring optimal performance.

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

  Answer:

  - blackscholes: we assign one thread to this job. Based on our experience, this job never utilizes more than one core and has a relatively short makespan, so additional threads are unnecessary.
  - canneal: we assign one thread to this job. With a short runtime and an early position in the queue, it never uses two cores. Thus, using more than one thread would have just implied extra overhead.
  - dedup: we assign one thread to it. This job has a very short makespan, so no speedup over the base performance with 1 thread/1 core is required.
  - ferret: we assign one thread to it. Although this job scales linearly from 1 to 2 threads (from Part 2b of the project), its queue position means that, in practice, it never gets two cores, making extra threads pointless.
  - freqmine: we assign two threads to it. This is the longest-running job of the bunch, and we need all the speedup we can get. We gave this job two threads as it scales very well and has a long runtime. Further, because of its placement, this is the only job that can take advantage of the full two cores it will be assigned to.
  - radix: we assign one thread to it. This is the only job being run on the same core as memcached, and as such, it will always be run on one core, and it will be paused/unpaused very often. Thus, using multiple threads would serve no purpose, and it would instead hurt the performance by introducing extra overhead.
  - vips: we assign one thread to it. This job has a very short runtime and it will likely be placed on a single core by our scheduling policy. Although the job scales very well, the relatively short runtime and its position in the queue imply that it won't benefit from running with more than one core, as it would only add extra overhead.

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

  Answer:

  `memcached` is running concurrently with all other jobs. `radix` is sequentially colocated on a shared core with `memcached` and is possibly running concurrently to all jobs, but this depends on its runtime. `ferret`, `dedup`, `vips`, `blackscholes`, `canneal`, `freqmine` are all placed in the *high-priority* queue, and they can be colocated and run concurrently with each other in various configurations, depending on the time each job takes to run and which core they get dynamically assigned to. In our measurements we found that the same pattern was repeated throughout all the runs, so in the following bullet list we present the concurrency/colocated information based on the data from the runs we measured:

- `radix` is always running concurrently to `dedup`, `vips`, `blackscholes`, `canneal`, and, only in the runs we did with `qps_interval=1.75` also `freqmine`. As it is colocated on core 1, it is colocated with `memcached`, although they never execute on this core at the same time.
- `dedup` is run concurrently with `ferret`, `radix` and `memcached`. The job has been started on core 3, thus being colocated sequentially on the same core with `vips`, `blackscholes`, `canneal`, and `freqmine` once it gets upgraded to two cores.
- `ferret` is run concurrently with `dedup`, `radix`, `vips`, `blackscholes`, `canneal` and `memcached`. It is started on core 2, thus being colocated sequentially just with `freqmine`.
- `vips` is run concurrently with `radix`, `ferret` and `memcached`. It is started on core 3, thus being colocated on the same core with `dedup`, `blackscholes`, `canneal`, and `freqmine` once it gets upgraded to two cores.
- `blackscholes` is run concurrently with `radix`, `ferret` and `memcached`. It is started on core 3, thus being colocated on the same core with `dedup`, `vips`, `canneal`, and `freqmine` once it gets upgraded to two cores.
- `canneal` is run concurrently with `radix`, `ferret` and `memcached`. It is started on core 3, thus being colocated on the same core with `dedup`, `vips`, `blackscholes`, and `freqmine` once it gets upgraded to two cores.
- `freqmine` is run concurrently with `canneal` and `memcached` and sometimes `radix`. It is started on core 2, thus being colocated just with `ferret`.

The ordering of the jobs was carefully crafted to achieve the following goals:

- Run `ferret` and `freqmine` as first and last job (to avoid interference). `freqmine` is left last so that it can use two cores and finish early.
- To enforce the previous point's goal, we placed as many jobs as possible sequentially running on the two cores of the *high proprity* queue in between `ferret` and `freqmine`. This way, `ferret` has enough time to complete before `freqmine` starts.
- Jobs that run concurrently on different cores, mainly the ones running during `ferret`'s runtime, needed to have minimal interference on the `llc` and `membw`, in order to avoid slowing down `ferret`'s execution.

Colocating jobs concurrently on a single core was considered but ultimately avoided, as we are forced to run `ferret` and `freqmine` sequentially, in order to avoid massive interferences. Therefore, even if we were successful in finding a concurrent configuration that would lower the makespan for other jobs, the overall runtime would still be lowerbounded by the execution of `freqmine` and `ferret`.

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

  Order:

  (a) *low-priority* queue: radix.

  (b) *high-priority* queue: ferret, dedup, vips, blackscholes, canneal, freqmine.

  Why:

  The main objective of the scheduler is to reduce runtime while maintaining the SLO.

For the first goal, we aimed to colocate other jobs on the `memcached` cores (those in the *low-priority* queue). Our choice was `radix`, which has very little interference and a very short runtime. To make the most out of the remaining resources, we had to find the sequence of jobs that would cause the least amount of interference, yielding minimal runtime and SLO violations. The major issue was the significant interference between `ferret` and `freqmine`, so they were placed first and last in our order. In particular, `freqmine` was placed last to scale to two cores, significantly decreasing its runtime. The first job in the *low-priority* queue and the first two jobs in the *high-priority* queue are started immediately.

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

  Answer:

  Part three was mainly different for two reasons:

  (a) We had more machines to work with, which allowed us to easily schedule `memcached` in a way that would not break the SLO.

  (b) The scheduling was to be defined statically and we had much less granular control on the start/stop times of jobs. Further, we also didn't have the option to pause/unpause their execution.

  Thus, in Part 3, we quickly discovered that placing `memcached` on the smallest node, either alone or with a job exhibiting low interference, easily satisfied the SLO requirements. However, in the Part 4 scenario, with only one machine available, avoiding SLO violations became significantly more challenging. This required careful consideration of interferences and some threshold tuning.

  While in Part 3, determining an optimal arrangement of jobs to place on machines not running `memcached` was sufficient, the new scenario necessitated maximizing CPU utilization to complete the scheduling quickly, which also required lowering the amount of resources allocated to `memcached` dynamically. Consequently, we also had to identify methods for colocating jobs on the same cores as `memcached`, while still maintaining the SLO. To achieve this, we had to monitor both incoming requests and CPU usage to identify suitable intervals during which the system load was sufficiently low to allow another job to run on a core previosuly reserved for `memcached`.

  Finally, the system's load is highly variable and significantly influences our scheduling decisions. As a result, the policy is highly dynamic, with very few execution parameters hardcoded, in contrast to the previous scheduling approach. For instance, the number of CPU cores assigned to a job, the total number of cores assigned to a job, and even the execution order are all subject to change based on system load and the performance of other jobs. Only a few parameters remain fixed, such as the number of threads for `memcached` and all jobs, and the initial job order in the queue. That said, even the same static queue order could result in different orderings at runtime, as we dynamically adjust the policy based on how quickly jobs terminate. Consequently, this scheduler is much more adaptable than the one proposed in Part 3, enabling it to respond effectively to various scenarios, including fluctuations in overall machine performance and sudden peaks in incoming requests.

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

Answer:

We used mainly these techniques to enforce our scheduling policy:

(a) Docker `cpuset`: to limit the cores available to a given job. This was both used at container creation time, as well as at runtime, to update a job's cores. For example, when there's only one job left in the *high priority queue*, it can take advantage of both available cores for that queue, and thus we would use the `update_container` method.

(b) `psutil.cpu_affinity`: to force `memcached` to run on a specific set of cores. The number of cores alternates between one and two, based on the QPS load and the CPU load. Thus, we use `set_affinity` to assign `memcached` to either cores `[0]` (on low loads), or `[0,1]` (on high loads).

(c) Docker `cpu_period/cpu_quota` are used to force the job not to use more than $\frac{\texttt{cpu\_quota}}{\texttt{cpu\_period}}\%$ of the CPU cycles. This did help preventing SLO violations and made the scheduler more stable. In our scheduler, we pose a strict $\frac{45}{200}\%$ limit on the CPU usage of the two memcached cores when we colocate another job. This constraint helped keep the latency low while colocating. Consequently, this docker option helped us maintain that constraint for longer periods of time, thus alleviating the high frequency switching in the job execution, that directly comes from the strict limit on the CPU usage.

(d) Data gathering: we gather two metrics from the system, which we use to enforce the scheduling. The CPU usage of cores `[0,1]` is collected every 100ms, and it is used to decide when a job can be colocated with `memcached`. Further, we read the incoming QPS directly from `memcached`, by connecting via `telnet` to port 11211 and issuing the `stats` command. This gives us the number of reads performed since the start of the `memcached` server. By keeping track of the previous readings, we implemented a sliding window that stores the incoming number of `read` requests. This sliding window is read at different widths to determine the stability of the system, yielding a very quickly converging algorithm to determine if the number of requests is changing (unstable) or stable.

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

Answer:

The core logic of our scheduler has been detailed previously. One crucial component that was not defined earlier is the `AugmentedQueue`. This is a straightforward data structure consisting of the following triple (Q, R, n), where:

(a) Q is a queue data structure, holding jobs which still need to be started.

(b) R is a list of length n, containing jobs that have been started but not yet completed.

(c) n specifies the number of jobs which can be executed concurrently, by placing each on a separate core. Consequently, n is equal to the number of cores managed by this `AugmentedQueue`.

In the following sections, we provide an overview of the implementation details that bring the aforementioned policy to life. The scheduler comprises four main parts:

(a) The `CPUReader` operates on a separate thread, polling CPU usage at an interval of 100 milliseconds. Reducing this interval to achieve higher resolution is not recommended, as indicated in the documentation of the `psutil` package. The collected data is kept in a sliding window. This information is subsequently aggregated by the `Scheduler` thread to inform decisions regarding the pausing or unpausing of jobs in the *low-priority* queue.

(b) The `QPSReader` operates on a separate thread that collects QPS readings from `memcached` by establishing a `telnet` connection to the server. This thread also maintains a sliding window of the collected values, enabling the `StabilityScanner` to access different window widths. Comparison of readings with different widths is employed to reliably and promptly determine the system's state.

(c) The `StabilityScanner`, which runs on the main thread, collects and aggregates data from the `QPSReader` to assess the system's stability. Our policy for determining a stable system is as follows:

$$\frac{|curr - qps|}{max(curr, qps)} > 30\%$$

The left-hand side of the expression calculates the percentage difference between the query readings from the last second (qps) and those from the last 400 milliseconds ($curr$). If this percentage difference exceeds 30%, the system is deemed unstable, and the scheduler is immediately alerted to prepare for a significant increase or decrease in the number of requests.

This check is performed every 75 milliseconds to ensure a prompt response to sudden changes. Additionally, when the system is considered stable, the scheduler is notified of the current QPS rate (computed over a sliding window of one second) every 140 milliseconds. This frequent notification is essential because the above criterion is not sensitive enough to detect small increases, such as 5K requests. This insensitivity is intentional; we aim to consider the system unstable only when there is a substantial change in the load, as handling minor changes less promptly is sufficient. Avoiding periods of instability is desirable, as they compel the scheduler to adopt a more conservative approach.

(d) Finally, the last component of the system is the `Scheduler`, which aggregates the data from the `StabilityScanner` and the `CPUReader` every 100ms, in order to apply the appropriate decisions, as dictated by the policy specified in Algorithm 1 and Part 4.2, question 1.

Algorithm 1 Scheduler policy: this algorithm shows the pseudocode of our policy. In this snippet, *loq_queue* refers to the *low-priority* `AugmentedQueue`, while *high_queue* referes to the *high-priority* `AugmentedQueue`.

while $|low\_queue.Q| \neq \emptyset$ **or** $|low\_queue.R| \neq \emptyset$ **or** $|high\_queue.Q| \neq \emptyset$ **or** $|high\_queue.R| \neq \emptyset$ **do**
  SLEEP(0.1) ▷ Sleep for 100ms
  $is\_stable, qps \leftarrow$ `StabilityScanner` ▷ Read stability state and incoming QPS
  $cpu_0, cpu_1 \leftarrow$ `CPUReader` ▷ Read CPU usage percentage
  if is_stable **and** $qps < 29K$ **and** $cpu_0 + cpu_1 < 45$ **and** $|low\_queue.R| > 0$ **then**
    SETCORES(*memcached*, $\{0\}$)
    UNPAUSEALL(*low_queue*)
  else
    PAUSEALL(*low_queue*)
    SETCORES(*memcached*, $\{0, 1\}$)
  end if
  $done\_low \leftarrow$ DONE(*low_queue*) ▷ Removes any finished jobs from R
  $done\_high \leftarrow$ DONE(*high_queue*)
  if $low\_queue.Q \neq \emptyset$ **and** $|low\_queue.R| < low\_queue.n$ **then**
    FILL(*low_queue*) ▷ Moves as many jobs as possible from Q to R and starts them
  end if
  if $high\_queue.Q \neq \emptyset$ **and** $|high\_queue.R| < high\_queue.n$ **then**
    FILL(*high_queue*) ▷ Moves as many jobs as possible from Q to R and starts them
  end if
  if $|high\_queue.R| < high\_queue.n$ **then** ▷ If this is true, then *high_queue* has no more jobs
    while $|low\_queue.Q \neq \emptyset|$ `or` $|low\_queue.R \neq \emptyset|$ **do**
      $job \leftarrow$ POP(*loq_queue*) ▷ Removes one element from either Q or R
      INSERT(*high_queue*, *job*)
    end while
  end if
  if $1 \leq |high\_queue.R| < high\_queue.n$ **then** ▷ There's a job running, not all cores are used
    $job \leftarrow high\_queue.R[0]$
    UPGRADE(*high_queue*, *job*) ▷ Give as many cores as possible to the first running job
  end if
end while

We also considered moving `dedup` to the *low-priority* queue, but ultimately decided against it as it did not result in any significant runtime improvement. We attribute this to the very short makespan of `dedup` and the limited time it can effectively execute while colocated on the same core as `memcached`. Therefore, we opted to avoid this approach to maintain cleaner plots and, most importantly, to ensure that two full cores remain available for `memcached` once `radix` completes. Nonetheless, the capability for queue movement has been retained and could be employed on a different machine or with a different workload if it proves beneficial.

Finally, we also considered an alternative policy: using concurrency for jobs that do not exhibit strong interference with each other, while running the remaining jobs sequentially, in addition to colocating one job with `memcached`, as we have done. We decided against this approach as it would have been more challenging to implement and likely less reliable due to the nature of interferences. Additionally, we do not believe it would have significantly improved the makespan, as the primary contributors to the current makespan are `ferret` and `freqmine`, which would need to be run sequentially in any case to avoid major interferences.

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 [9] across three runs. Compute the mean and standard deviation of the total time to complete all jobs - the makespan of all jobs. Fill in the table below. Also, compute the SLO violation ratio for memcached for each of the three runs; the number of data points with 95th percentile latency > 1ms, as a fraction of the total number of datapoints. The SLO violation ratio should be calculated during the time from when the first batch-job-container starts running to when the last batch-job-container stops running.

Answer:

job name	mean time [s]	std [s]
blackscholes	94.33	0.69
canneal	247.35	9.05
dedup	38.04	0.12
ferret	372.39	4.63
freqmine	300.75	2.02
radix	50.73	6.64
vips	97.69	2.24
total time	673.46	4.58

The following table shows the SLO violation ratios for each of the three runs:

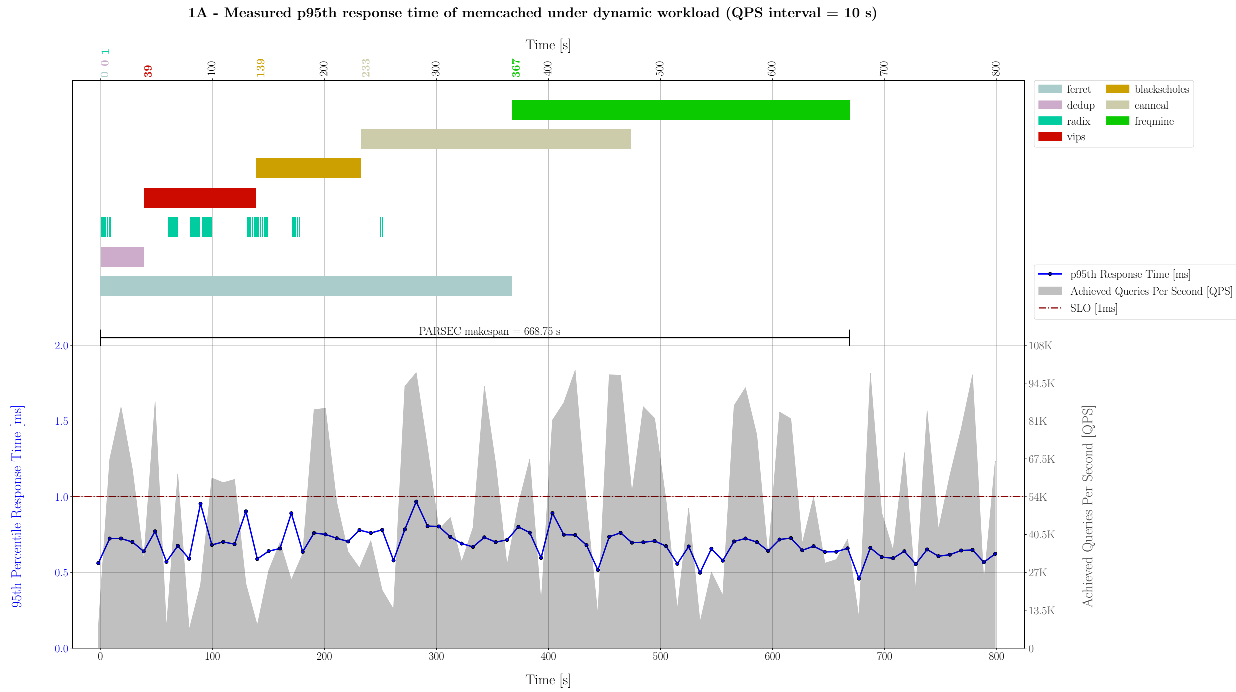
Run 1	Run 2	Run 3
$\frac{0}{66}$ (0%)	$\frac{0}{66}$ (0%)	$\frac{0}{66}$ (0%)

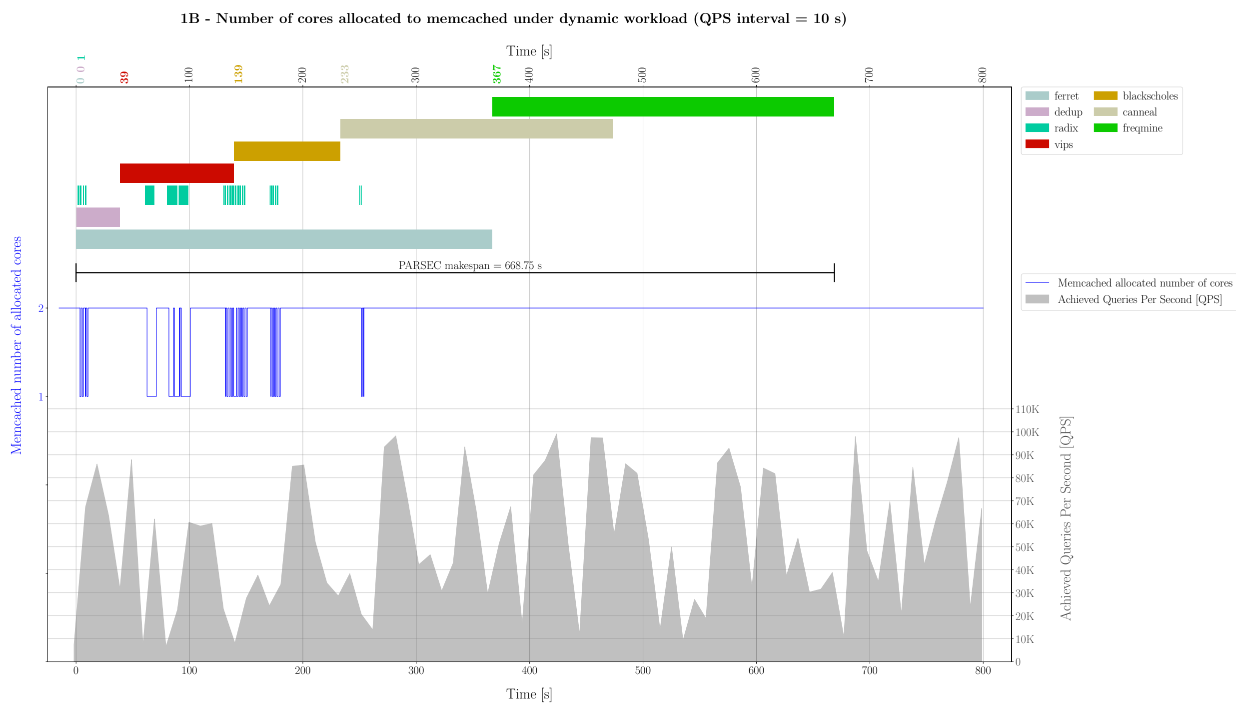
Include six plots – two plots for each of the three runs – with the following information. Label the plots as 1A, 1B, 2A, 2B, 3A, and 3B where the number indicates the run and the letter indicates the type of plot (A or B), which we describe below. In all plots, time will be on the x-axis and you should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

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

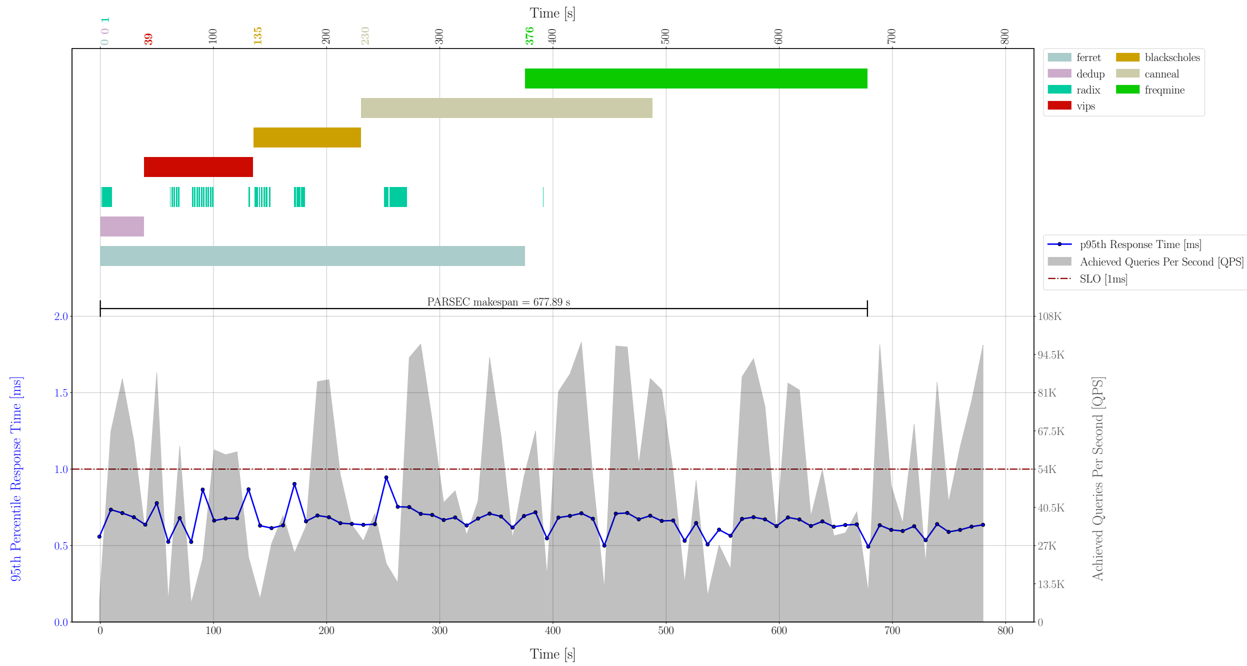
starts executing at which time. If you pause/unpause any workloads as part of your policy, you should also indicate the timestamps at which jobs are paused and unpaused. All the plots will have have two y-axes. The right y-axis will be QPS. For Plots A, the left y-axis will be the 95th percentile latency. 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`).

Plots:

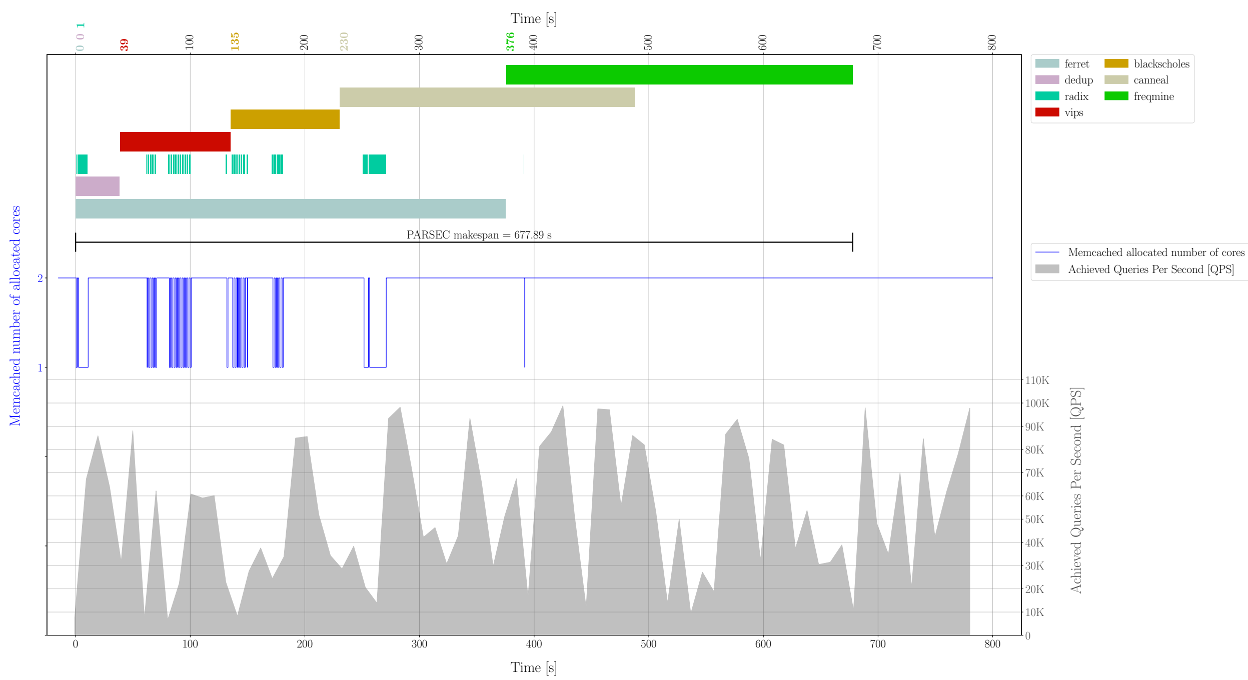




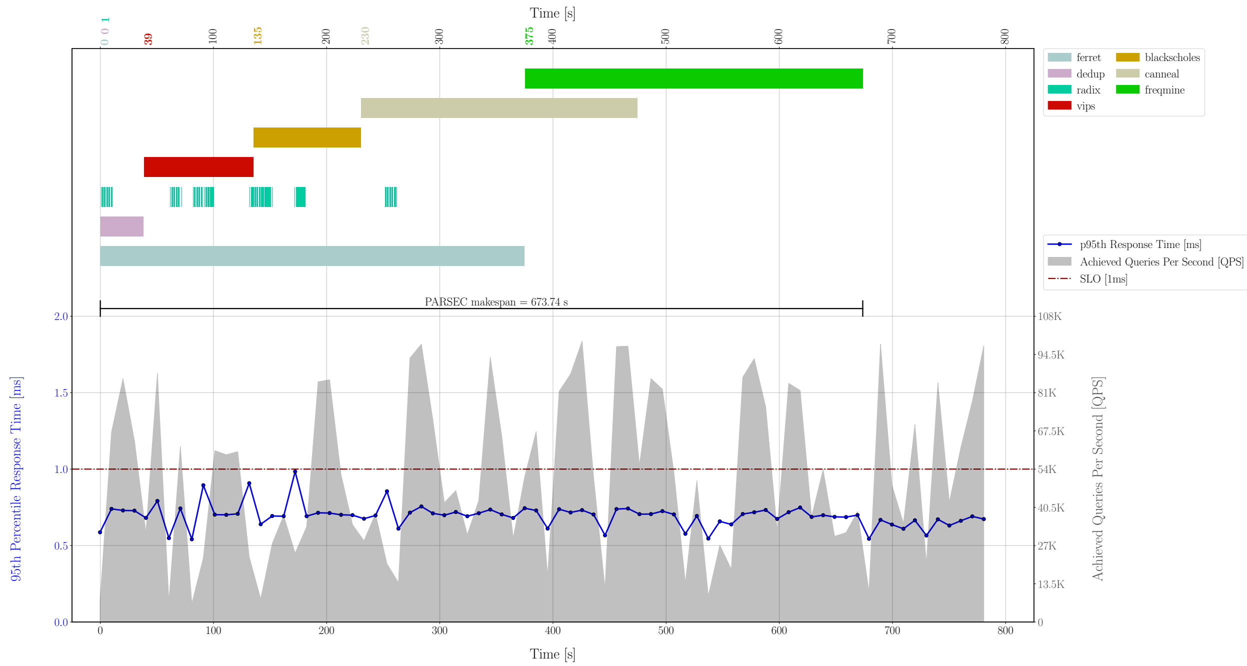
2A - Measured p95th response time of memcached under dynamic workload (QPS interval = 10 s)



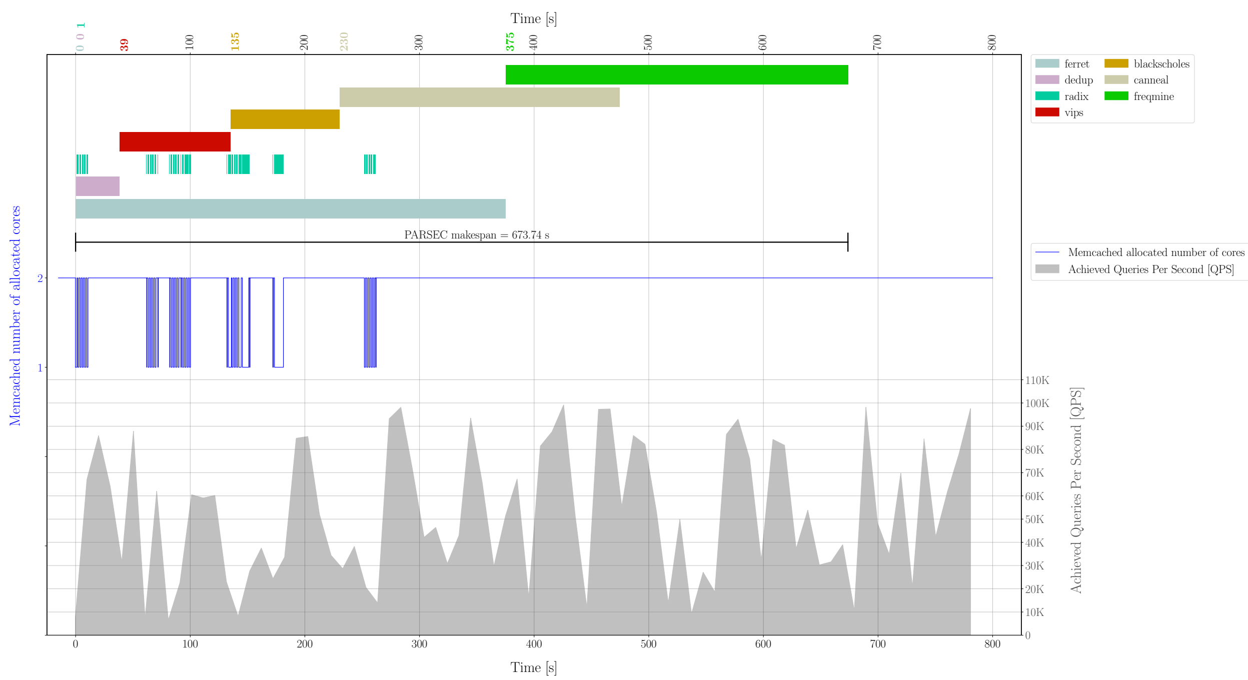
2B - Number of cores allocated to memcached under dynamic workload (QPS interval = 10 s)



3A - Measured p95th response time of memcached under dynamic workload (QPS interval = 10 s)



3B - Number of cores allocated to memcached under dynamic workload (QPS interval = 10 s)



Note: we included a mirrored x-axis at the top of each plot to improve readability. As requested, we annotated this axis with the starting times of each batch job. Each job is represented by a horizontal bar, indicating its execution period, with gaps denoting pause spans. As can be observed from the plots, only radix is paused/unpaused, and very frequently. Including precise timestamps for these pauses and unpauses would have cluttered the x-axis, making it unreadable. Therefore, our solution of using segmented horizontal bars aims to

maximize plot readability while adhering to the specified plot requirements.
We ensured proper synchronization between the batch jobs and `mcperf` data; the x-axis aligns with the start time of the first batch job's container. In particular, `memcached` and `mcperf` run before the first batch job starts and continue after the last batch job ends.
Moreover, as required by the handout, we ensured that `memcached` ended at least one minute after the scheduler.
We also included a black line indicating the obtained makespan.

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: To enhance the reliability of the results, we ran the command 3 times. Our measurements show that the policy with `qps_interval`=5s performs worse than the previous one with respect to the mean makespan obtained. While the SLO violation ratios of the three runs were still 0% ($\frac{0}{138}$, $\frac{0}{135}$ and $\frac{0}{138}$), the mean makespan increased to 675.22 (1.76s increase), with a standard deviation rising to 6.46s (1.88s increase). From `qps_interval=10` to `qps_interval=5s`, most batch jobs' mean times varied slightly (variations in the range [0.53s - 1.03s]), except for `blackscholes` and `ferret`, which saw a 2.13s increase and a 3.03s increase, respectively. Standard deviations for batch jobs also varied slightly (variations in the range [0.01s - 1.4s], except for `blackscholes` and `freqmine`, which both increased by 3.19s.

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:
The following table shows the SLO violation ratios with `qps_interval`=5s for each of the three runs:

Run 1	Run 2	Run 3
$\frac{0}{138}$ (0%)	$\frac{0}{135}$ (0%)	$\frac{0}{138}$ (0%)

What is the smallest `qps_interval` you can use in the load trace that allows your controller to respond fast enough to keep the memcached SLO violation ratio under 3%?

Answer:
The smallest qps_interval we could employ to achieve a SLO violation ratio under 3% is qps_interval=1.75s.

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

Answer: The rationale for selecting this value is partly attributable to the operational mechanism of our StabilityScanner: it compares the readings from the past 1 second and 400 milliseconds within the sliding window to determine if there is a significant disparity. This indicates that, in scenarios where the request rate is rapidly fluctuating, we require approximately 400 milliseconds of sustained increased load to detect any change. Furthermore, when the load changes, it does so gradually, as evidenced by a series of consecutive measurements. Therefore, we realistically need around 600 milliseconds to accurately flag the system as unstable. Finally, this comparison only triggers an unstable system state when the discrepancy is significant, thus we may ignore minor changes in the load and be even slower to react.

Secondly, the logic of our scheduler also influences this choice. Upon stabilization of the system, we utilize the QPS reading to provide a reliable metric. This involves averaging the measurements from the past second, which can lower the actual current number of incoming requests during periods of rapid transitions from a low load state to a high load state. To maintain system stability and avoid frequent pausing and unpausing of jobs in the low-priority queue, we may be slow to respond to minor changes (2.5K-5K QPS) in load, which could cause violations when the system is near its limits.

To ensure reliability, we selected a value that approximates the sum of the two delays in our system: 1 second for the QPS reading to stabilize and 600 milliseconds for instability detection. Consequently, we conducted our experiments with `qps_interval=1.75`.

We are confident that the approach in our scheduler is solid, and that this performance could be improved with some tweaking of the stability logic. In particular, going back, we would have increased the polling rate of the QPSReader, as well as use a smaller window to detect the instability. This would make our StabilityDetector trigger an unstable state more often, which would make the scheduler overall much more conservative. Further, we could also read a smaller part of the sliding window in the Scheduler, to use more up-to-date values for the policy enforcement. This wasn't done in the first place to minimize the runtime, but as we've noticed after the measurements, we could have paused radix more often and still achieved a comparable makespan. In conclusion, polling more frequently and being more conservative would most definitely allow us to achieve a 1 second or lower interval, but this path wasn't pursued due to a lack of time.

Use this qps_interval in the command above and collect results for three runs. Include the same types of plots (1A, 1B, 2A, 2B, 3A, 3B) and table as in Question 3.

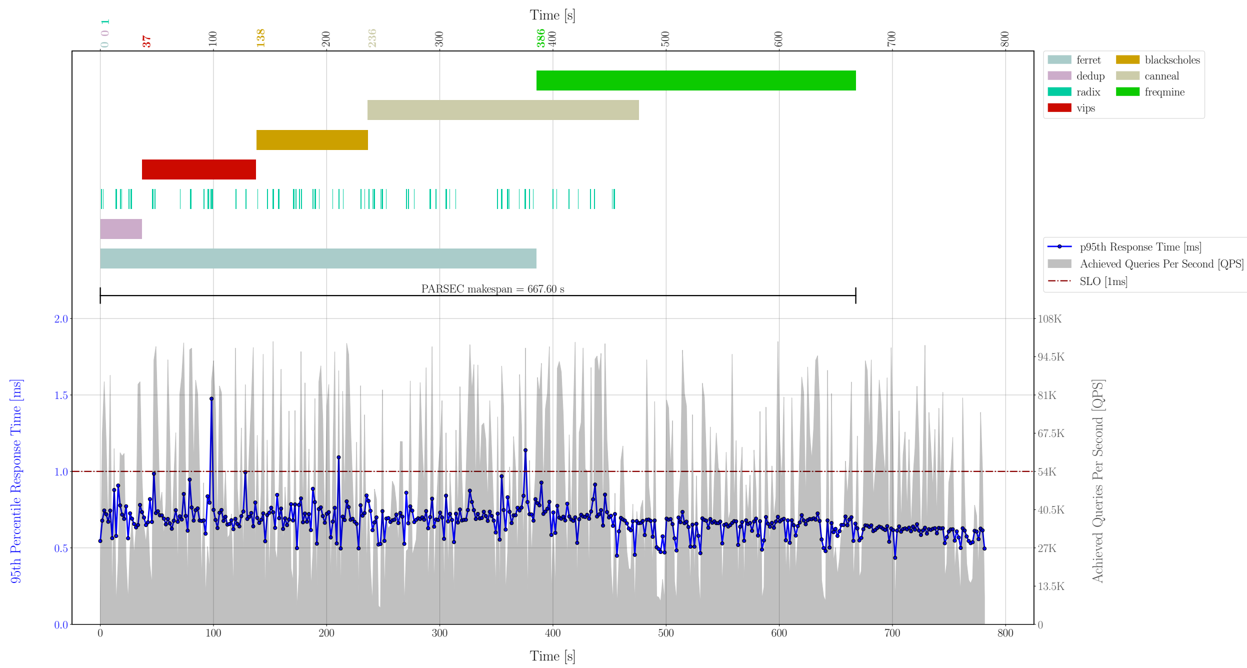
The following table shows the SLO violation ratios with qps_interval=1.75s for each of the three runs:

Run 1	Run 2	Run 3
$\frac{3}{380}$ (0.79%)	$\frac{7}{387}$ (1.81%)	$\frac{10}{389}$ (2.57%)

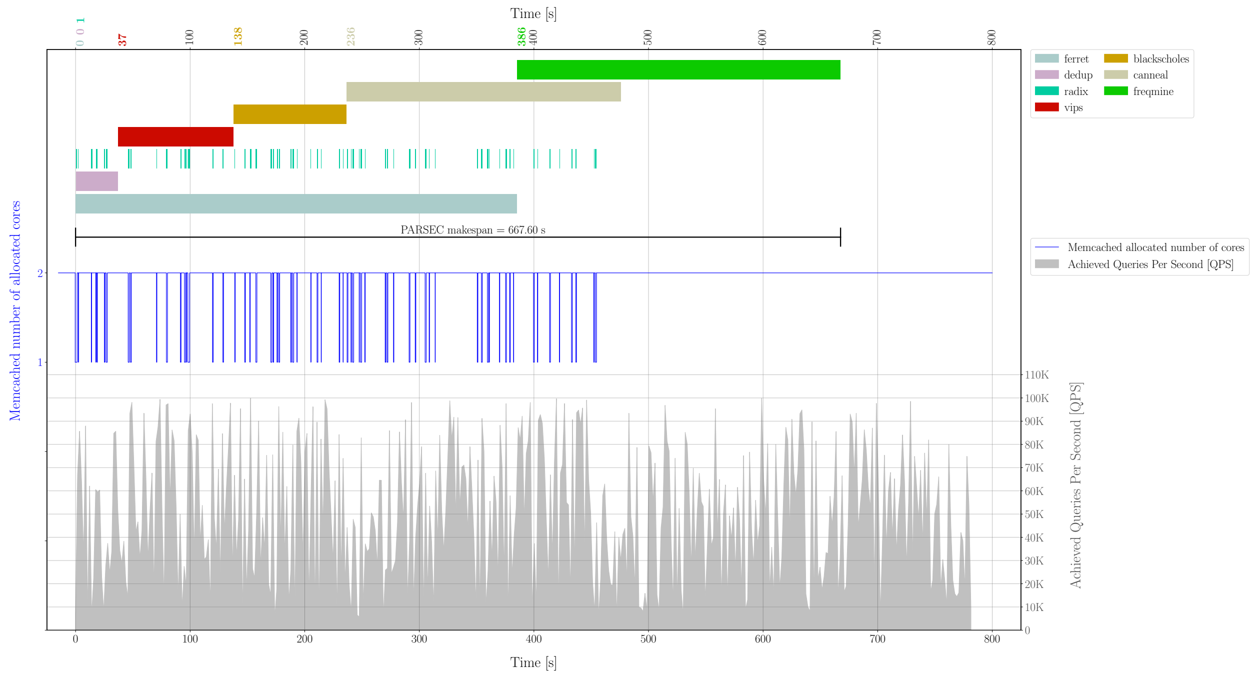
Plots:

job name	mean time [s]	std [s]
blackscholes	99.34	1.54
canneal	246.03	7.27
dedup	36.92	0.44
ferret	389.08	9.53
freqmine	287.02	7.66
radix	45.38	4.13
vips	101.15	0.80
total time	676.41	8.05

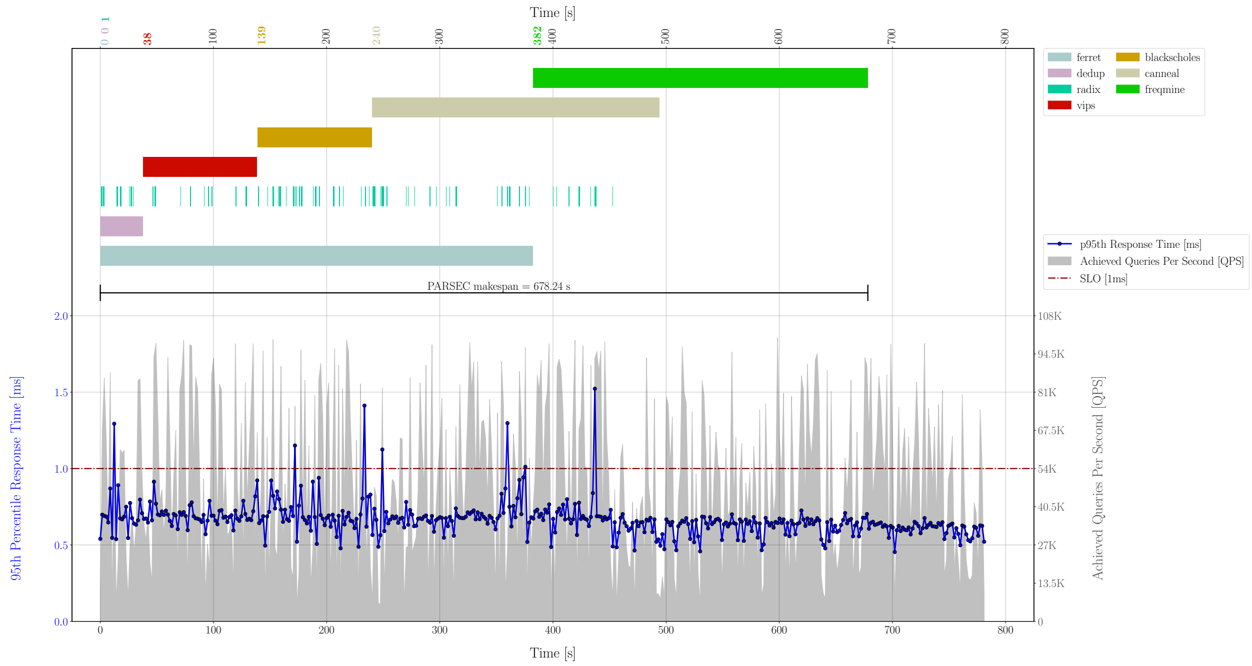
1A - Measured p95th response time of memcached under dynamic workload (QPS interval = 1.75 s)



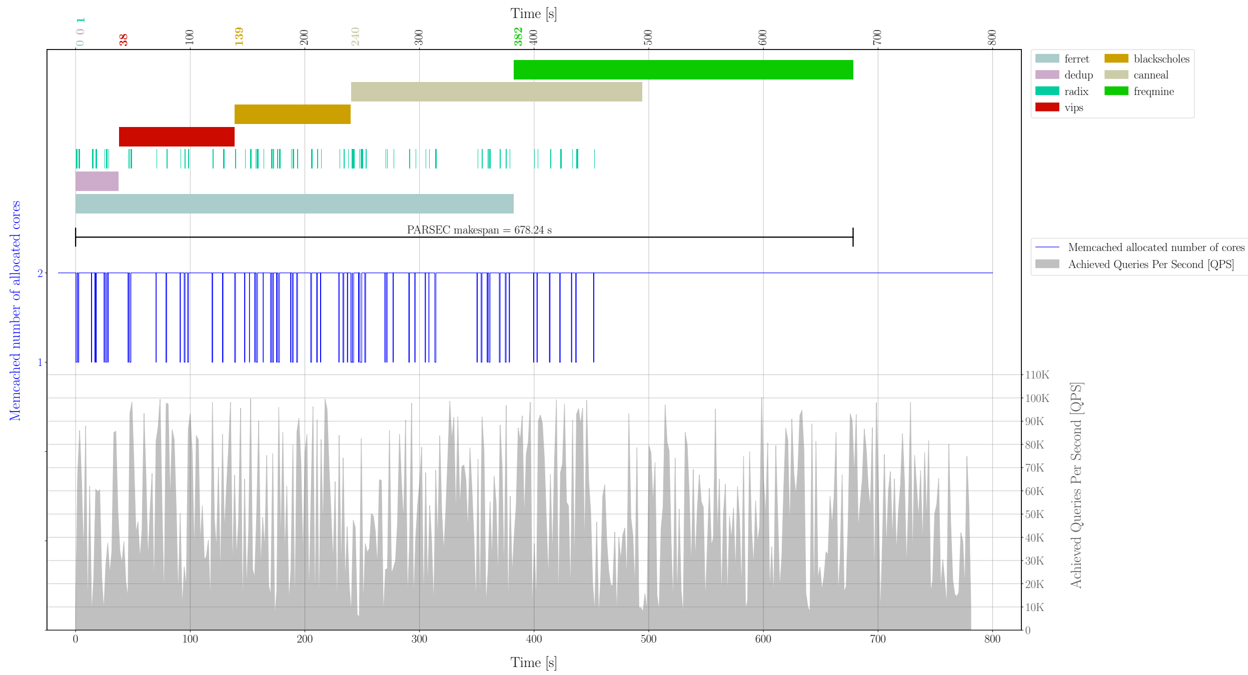
1B - Number of cores allocated to memcached under dynamic workload (QPS interval = 1.75 s)



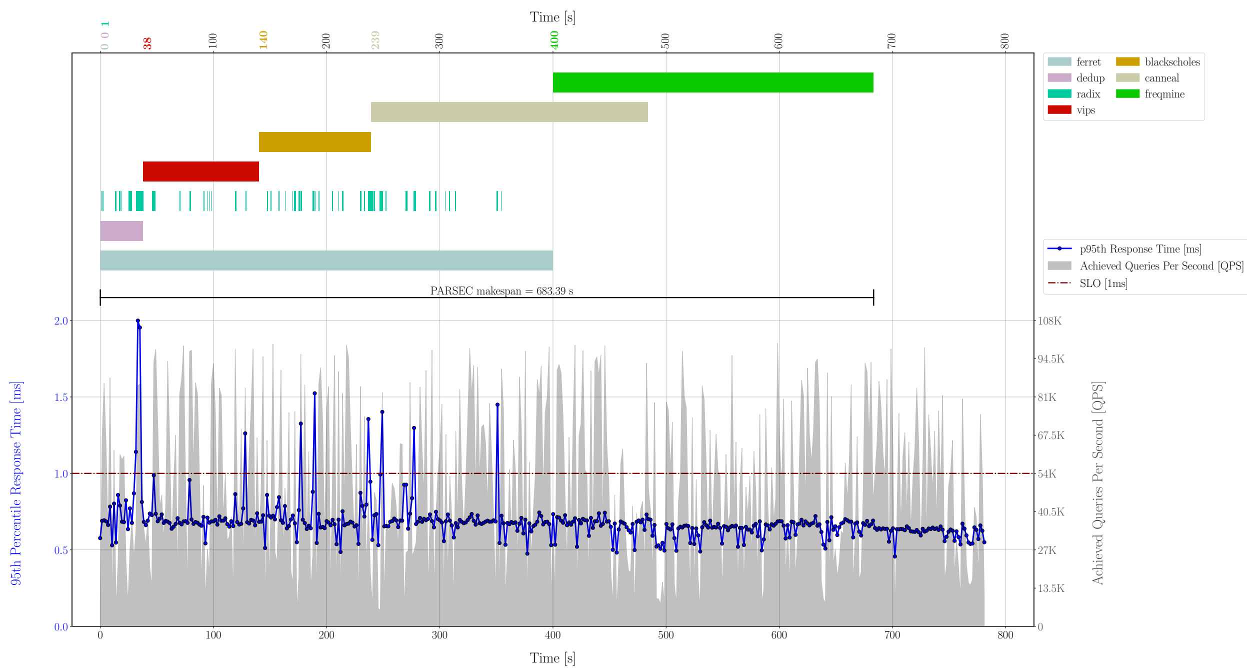
2A - Measured p95th response time of memcached under dynamic workload (QPS interval = 1.75 s)

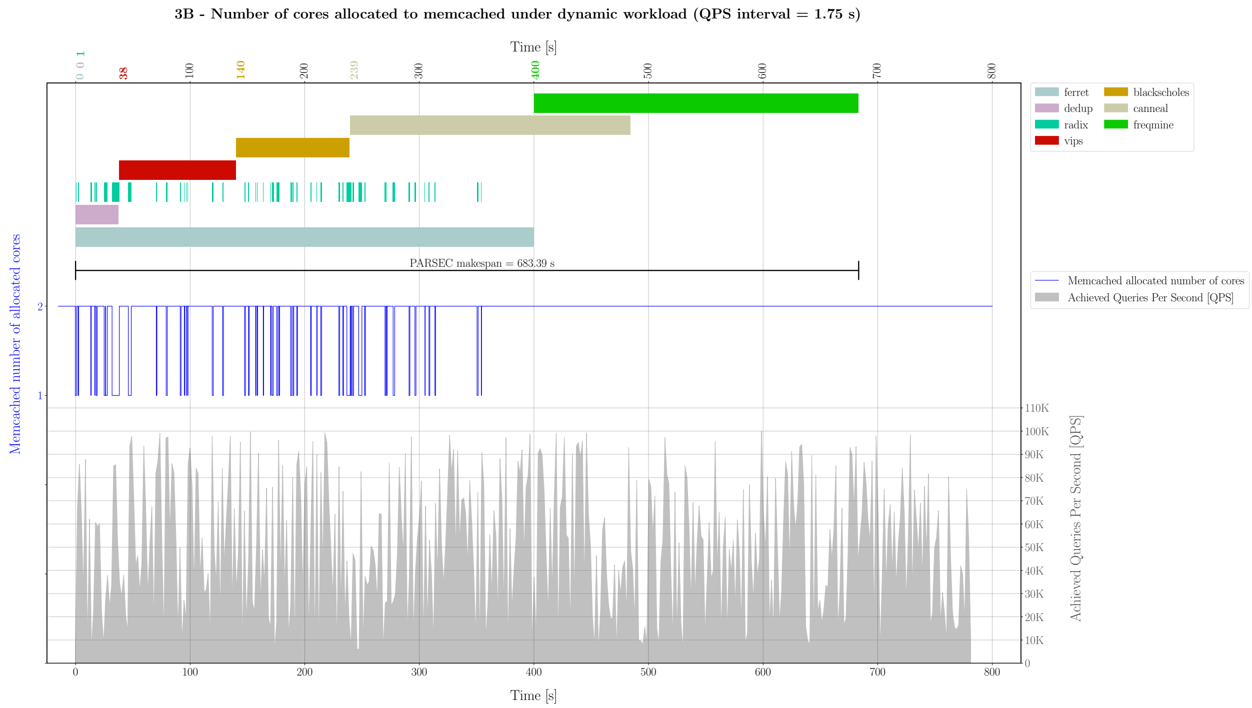


2B - Number of cores allocated to memcached under dynamic workload (QPS interval = 1.75 s)



3A - Measured p95th response time of memcached under dynamic workload (QPS interval = 1.75 s)





Note: The same general concepts explained for the other set of plots (Question 4.3) apply here as well.