Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Systems@ETH Zürich

Cloud Computing Architecture

Semester project report

Group 73
Josué Fleitas Sánchez - 24-935-587
Amin Oudrhiri - 24-953-853
Damian Herculano - 21-818-083

Systems Group
Department of Computer Science
ETH Zurich
May 16, 2025

Instructions

- **Please do not modify the template!** Except for adding your solutions in the labeled places, and inputting your group number, names and legi-NR on the title page.

  Divergence from the template can lead to subtraction of points on graded parts of the project.

- Parts 1 and 2 of the project are **ungraded**. They will help you analyze the behavior of the applications you will have to run on parts 3 and 4 (graded), for which you will have to design a scheduling policy **based on the information** you will gather on **parts 1 and 2**.

- Be conservative with your credits! Parts 3 and 4 might need extra credits for experimentation. Parts 1 and 2 should cost you approximately **15 USD**.

- **Use of AI Tools**: The use of AI tools must adhere to ETH regulations. **You take responsibility for the content you submit**. You must disclose any use of GenAI and other AI tools in your work and avoid sharing copyrighted, private, or confidential information with commercial AI platforms. Failure to comply with these guidelines may result in disciplinary action.

Contributions

1. **Amin Oudrhiri**: setup work, scheduler designs, coding/scripting, profiling experiments and report writing (Parts 1, 2, 3 and 4)

2. **Josué Fleitas**: setup work, scheduler designs, coding/scripting, profiling experiments and report writing (Parts 1, 2, 3 and 4)

3. **Damian Herculano**: setup work and report writing (Part 2, Q2)

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	121.7	0.57
canneal	159	1.0
dedup	48.7	3.05
ferret	165.33	0.58
freqmine	143.0	0
radix	14	0
vips	129.33	11.93
total time	166.1	1.0

Answer: Our SLO violation ratio for the three runs is 0% showing that we comfortably observe the SLO. One of the reasons we were able to achieve this was because we are isolating the memcached process on a single core, reducing the amount of critical L1i and CPU interferences that we observed in Part 1.

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: The plots showing the 95th percentile tail latencies for the memcached process across time during the scheduler's execution are shown in Figures 4, 5 and 6 below.

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

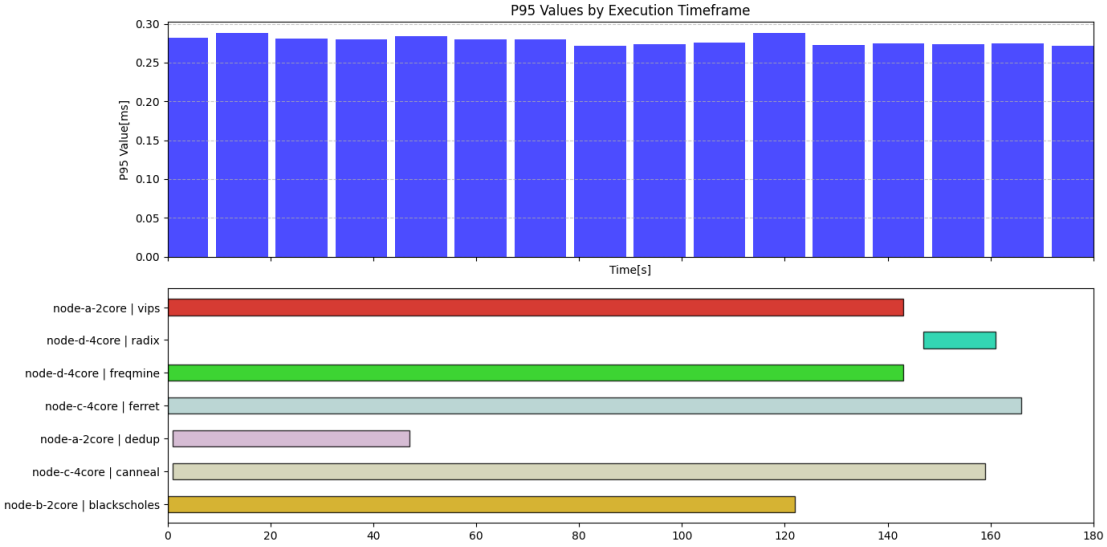

Figure 4: P95 Latency of Memcached Across Time (Run 1)

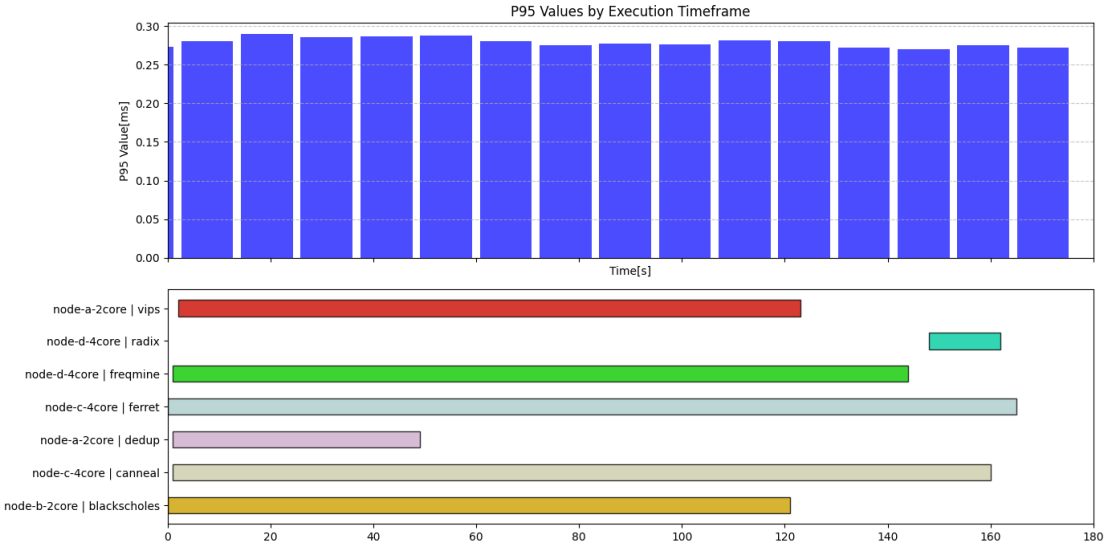

Figure 5: P95 Latency of Memcached Across Time (Run 2)

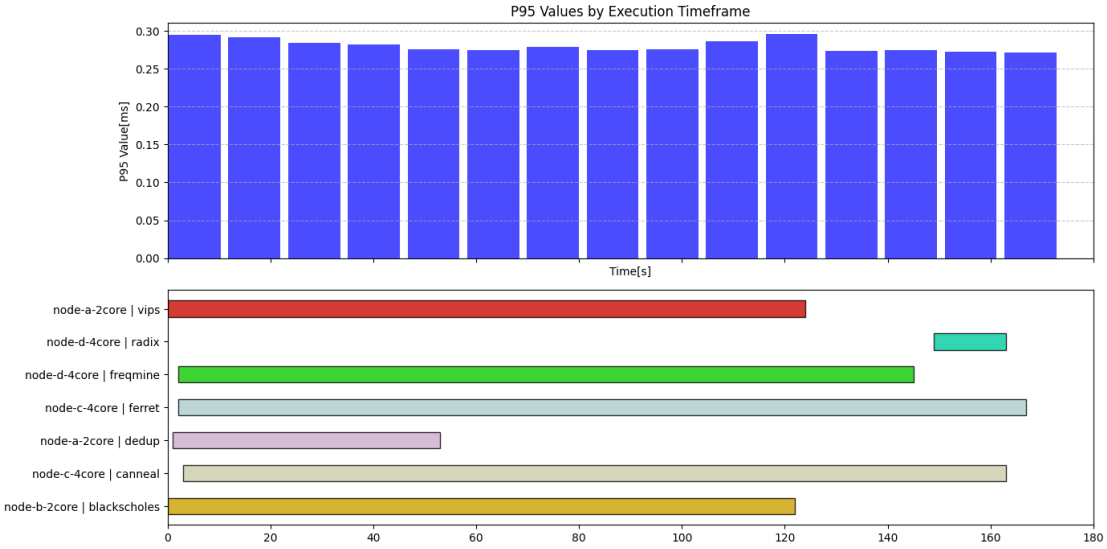

Figure 6: P95 Latency of Memcached Across Time (Run 3)

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

   - Which node does memcached run on? Why?
     Answer: Memcached is run on Node B because in part 1 we noticed that interferences related to memory bandwidth had little effect on its tail latencies, thus it made sense to run it on the VM with the lowest memory resources. Moreover, since the load we are running on is relatively low (30K QPS) and the process is significantly affected by L1i and CPU interference, we also knew the process could run on a single isolated core and likely still respect the SLO. Thus, we opted to pin the memcached process (using the taskset command) to one of the faster cores which narrowed the decision down to the Node B VM.

   - Which node does each of the 7 batch jobs run on? Why?
     Answer:
     - blackscholes: We run blackscholes on the other core of the Node B VM (pinning it using taskset). The reason for this is that we know blackscholes involves a large number of FLOP instructions [Bienia et al., 2008], so we needed it to run on a machine with a faster core. Blackscholes also has a fairly small working set compared to other workloads, meaning it can comfortable fit in node-b.
     - canneal: Canneal runs on the 4 core Node C VM since we noticed that it has a fairly large working set (1-2GB) that it doesn't fit on Node B Bienia et al. [2008]. We also observed a significant performance improvement of approximately 3x when scaling from 1 to 4 cores/4 threads in part 2. When considering the job's duration, running it on a 4-core configuration yields the best gain in terms of makespan reduction.
     - dedup: As one of the lighter-weight jobs, dedup is able to run on the slower 2 core Node A VM and can be collocated with other processes since its interference patterns (as seen in Part 2) are quite lenient. Since the job also involves a lot of reads/writes, it made further sense to choose the high-memory virtual machine for its execution.

- ferret: Ferret is run on the 4 core, high CPU Node C VM. The reason for this is that the instruction profile for ferret also reveals a large number of FLOPs and further benefits significantly from parallelism according to the results in part 2. Thus, running it on a faster VM with a larger number of threads made the most sense.
- freqmine: Freqmine is run isolated on the Node D standard VM. This is because freqmine is very sensitive to most forms of interferences and according to part 2 benefits greatly from parallel execution on multiple threads. Letting it fully utilize the 4 cores using a 4 thread configuration proved to reduce the makespan the most.
- radix: Radix is run following the completion of freqmine on the 4 core Node D VM. The main reason for this is that it has one of the largest working set (4GB) and would thus be best suited to run on one of the 16GB machines. Since we were looking to collocate quite a few jobs on node A we thus opted to place radix after freqmine in Node D. Since the job itself takes quite a short time to complete, this additional wait time could be absorbed into the runtime of some of the other benchmarks. Additionally, radix does not involve many FLOPs (since it is an integer sorting algorithm) so it would not have necessarily benefited from a faster CPU.
- vips: We run this job on the slower 2 core Node A VM alongside dedup. This decision boiled down to the instruction profile of vips Bienia et al. [2008], displaying an even balance between memory operations and cpu instructions, making it less optimal to run node-B (since that node has poor memory rescources). While it did benefit a lot from parallelism on 4 threads/4cores, we preferred placing longer jobs on the 4 core machines instead, leaving only node-A as the option.

- Which jobs run concurrently / are colocated? Why?
  Answer: We collocate the following jobs:

  - **Canneal and Ferret**: The interference profile in Part 2 shows that canneal is not greatly affected by interferences overall, making this benchmark a prime candidate for collocation. Furthermore, canneal involves very few FLOPs (low CPU usage) and has one of the lowest numbers of instructions out of all of the benchmarks [7.33B [Bienia et al., 2008]] and thus has low L1i cache usage. This suggested that some time could be saved if canneal was successfully collocated with either ferret or freqmine, as the interferences caused by it should not affect these two jobs by a significant amount. We then empirically decided that the best setting was to collocate it with Ferret.
  - **Vips and Dedup**: Although vips displays some sensitivity to interference, its execution remains largely isolated since dedup completes quickly, meaning that, in reality, they run concurrently for only a short period. Furthermore, any performance impact on dedup is essentially mitigated by its shorter duration, preventing these slowdowns from significantly affecting the overall makespan.

  .
- In which order did you run the 7 batch jobs? Why?
  Order (to be sorted): blackscholes, ferret, canneal, [freqmine, radix], vips, dedup
  Why: The order above represents the order in which we spun up the containers (which is close to negligible). The only real blocking/isolated process is freqmine whose inter-

ference profile is very sensitive to everything. Thus, radix does wait for freqmine to complete before beginning execution.

- How many threads have you used for each of the 7 batch jobs? Why?
  Answer:
  - blackscholes: we ran blackscholes on a single thread since we are pinning it to one of the 2 cores on Node B to avoid interferences with memcached. Adding more threads, although potentially benefitial given the parallel speedup observed in Part 2, would introduce a non-negligible context-switching overhead, and we thus avoid it.
  - canneal: 2 threads, based on the results in part 2 and the fact that its one of the more longer jobs, we would get the most overall gain by running it in a parallel fashion.The decision to run it on 2 threads over 4, boils down to reducing cpu-core contention and making it less interfere with ferret.
  - dedup: based on part 2, dedup showed the least speedup when parallelizing. Given that its also collocated to vips, we also wanted to make this process as lightweight as possible, making us opt for 1 thread.
  - ferret: 4 threads were assigned to this process. This is motivated by the parallel speedup profiling done in Part 2 revealing that the job could benefit signficantly from parallelism. Coupled with the fact that this is one of the longest-running jobs, it made sense to fully utilize the cores in the VM this is runnning on.
  - freqmine: by the same line of reasoning as ferret, 4 threads were chosen for this job. In other words, this is one of the longest-running jobs and coincidentally also exhibits a moderate speedup through parallelization.
  - radix: 8 threads. Radix was comfortably the benchmark with the largest parallel speedup and by setting the number of threads to this number, we were able to make its runtime negligible relative to the other jobs.
  - vips: we chose to run this job using 2 threads since we are running it on a 2-core machine but it still benefits from parallelization.
- Which files did you modify or add and in what way? Which Kubernetes features did you use?
  Answer: To implement the scheduler, we modified the YAML files for each of the benchmarks by changing the number of threads each program was running on (setting the appropriate flag in the run command) and by pinning the processes to specific cores if needed by prefixing the run command using the taskset command. We additionally experimented with using Kubernetes' own load-balancing and setting core requirements and limits but this proved to be less reliable in terms of final makespan than controlling the cores each benchmark was running on directly. Our scheduling logic was implemented in a simple Python script ("part3_schedule.py") which contains, among other things, the logic to block Kubernetes pods before executing others to enforce the order presented in one of the sections above.

- Describe the design choices, ideas and trade-offs you took into account while creating your scheduler (if not already mentioned above). Describe how your policy (and its performance) compares to another policy that you experimented with (in particular to the second-best policy that you designed).

Answer: We made the following design decisions:

- Minimizing number of free cores: the key guiding principle for the scheduler was to minimize the number of free cores at any given time. This is achieved by spreading long-running jobs across the VMs that we have access to, to avoid under-utilizing any of our resources.
- Isolating interference sensitive jobs: we incorporate the knowledge gained from the measuring done in Part 2 in order to reduce the potential resource interference between collocated jobs. In particular, we try to collocate jobs that are more lenient in terms of interferences, while considering the particular resources that each job make the most use of.
- Parallelizing longer jobs: although many of the runtimes can be shown to improve by leveraging parallelism, we prioritize parallelizing longer jobs (like freqmine and ferret) as the proportion of time saved is significantly higher than for shorter-running jobs. This helps us bring down the total makespan considerably more than if we prioritized parallelism based on the percentage speedup achieved by each job.
- Collocating shorter jobs: Collocating shorter jobs alongside jobs that are less sensitive to interferences enabled us to "absorb" the runtime of the shorter job within the runtime of the longer job. For some cases this was better than running the jobs sequentially, as the cost of the interferences were less than that of running the 2 jobs sequentially.

This policy gave us the best final makespan even though we tried a few other ideas that we abandoned in the end:

(a) Using CPU Core affinity on specific collocated jobs: while reducing collocations by pinning some of the jobs to separate cores helped to reduce the runtime for some jobs (in particular, separating freqmine and canneal to run on 2 isolated sets of 2 cores helped reduce the runtime for canneal), this always had a negative effect on the longer job. Since there is no easy way of dynamically setting the number of cores that a pod is running on other than through a manual SSH into the machine (to our knowledge), we opted to ditch this idea.

(b) Collocating a job on the same core as memcached: While this may have reduced the makespan at some times and give more core time to the rest of jobs, this ended up violating the SLO.

(c) Limiting the number of cores for jobs: We found that letting jobs fully utilize the resources of the VM they were placed in worked best for our strategy. Limiting core usage might have helped in some instances but we didn't find any that were beneficial at a global level.

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 220K in order to answer the following questions:

```
$ ./mcperf -s INTERNAL_MEMCACHED_IP --loadonly
$ ./mcperf -s INTERNAL_MEMCACHED_IP -a INTERNAL_AGENT_IP  \
           --noload -T 8 -C 8 -D 4 -Q 1000 -c 8 -t 5 \
           --scan 5000:220000:5000
```

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: Our plot showing the variation of QPS and 95th percentile tail latency of the memcached service is shown in Figure 7 below. The averaged values show the mean of three separate runs while the error bars show the standard deviation of the runs.

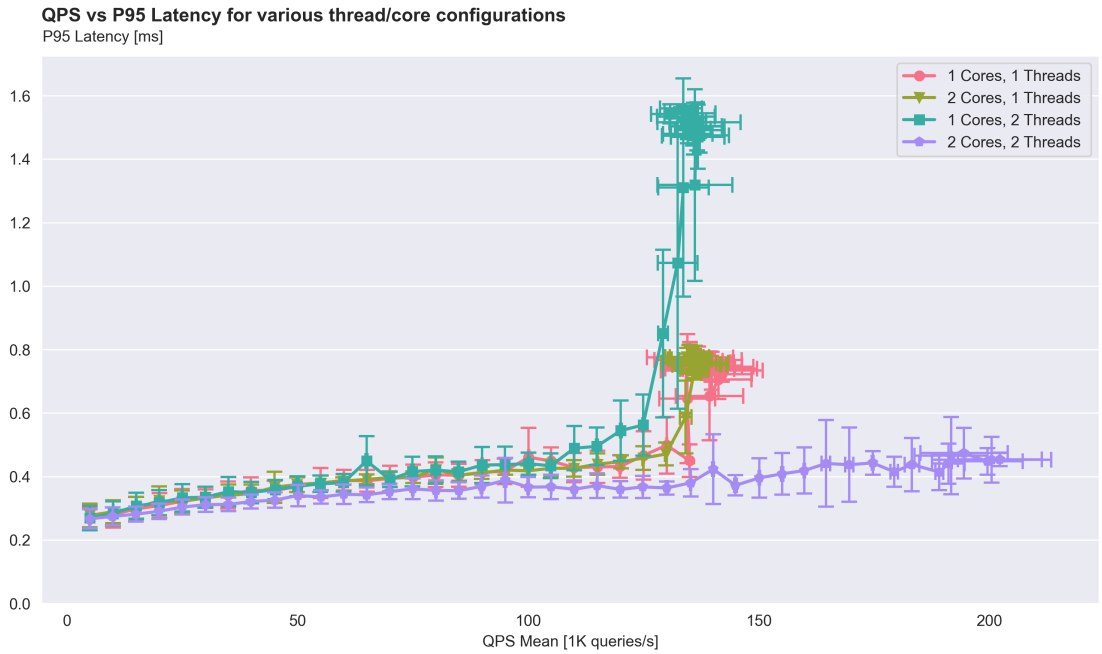

Figure 7: QPS vs P95 Latency of Memcached Service Running on Different Core and Thread Combinations

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: From the plot we conclude that in order to guarantee the SLO beyond the 120-130K QPS mark we need to be running the service on 2 cores and 2 threads. Importantly, this means that the service needs to be upscaled from running on a single core to running on multiple cores at this threshold as otherwise it saturates and risks violating the SLO.

b) [2 points] To support the highest load in the trace (around 220K QPS) without violating the 0.8ms latency SLO, how many memcached threads (T) and CPU cores (C) will you need?

Answer: We notice that beyond $\sim$ 140K QPS, all the confiugartions except $T = 2$ threads $C = 2$ cores violate the SLO. Thus we need $T = 2$ threads and $C = 2$ cores to support the highest load in the trace.

c) [1 point] Assume you can change the number of cores allocated to memcached dynamically as the QPS varies from 5K to 220K, 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 0.8ms 95th percentile latency SLO while the load varies between 5K to 220K QPS?

Answer: As seen in the plot we would use 2 threads in this scenario. To consistently meet the SLO beyond the 100K QPS region, memcached should run with a configuration of $T = 2$ threads and $C = 2$ cores to prevent saturation, as observed with other configurations. Conversely, when QPS falls below this region, the core count can be scaled down to $C = 1$ to avoid unnecessary resource usage.

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 220K.

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 230K. In each graph, use two y-axes. Plot the 95th percentile latency on the left y-axis. Draw a dotted horizontal line at the 0.8ms 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: The plots showing the P95 Latency vs CPU Utilization for the memcached process running on 2 threads and different number of cores are presented in Figures 8 and 9 below.

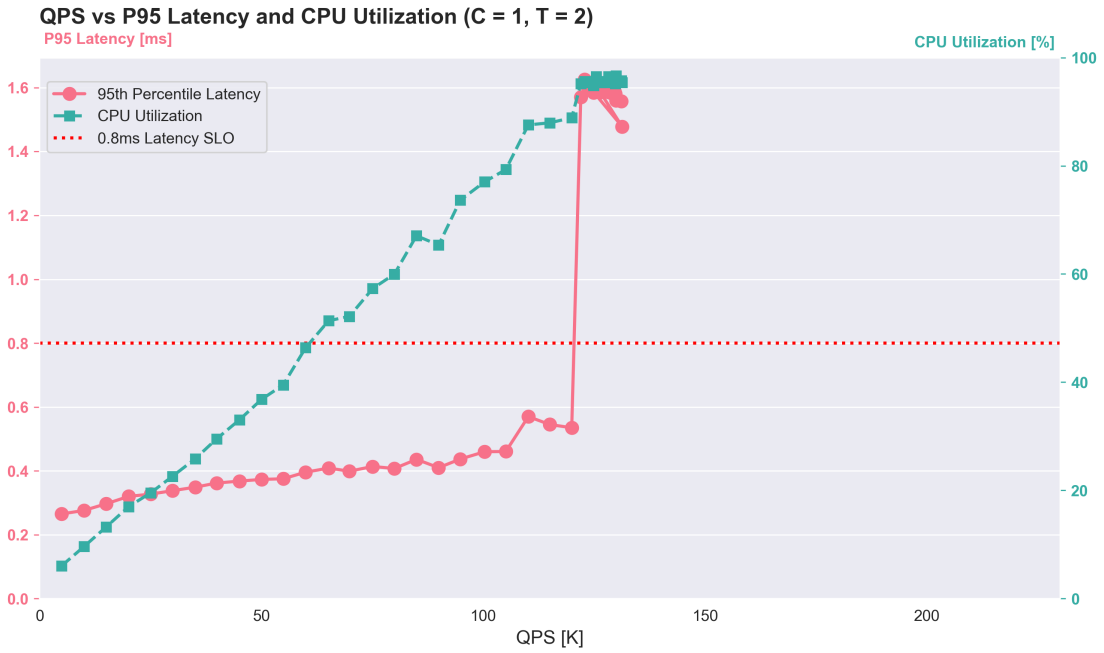

Figure 8: P95 Latency vs CPU Utilization for 2 Threads, 1 Core

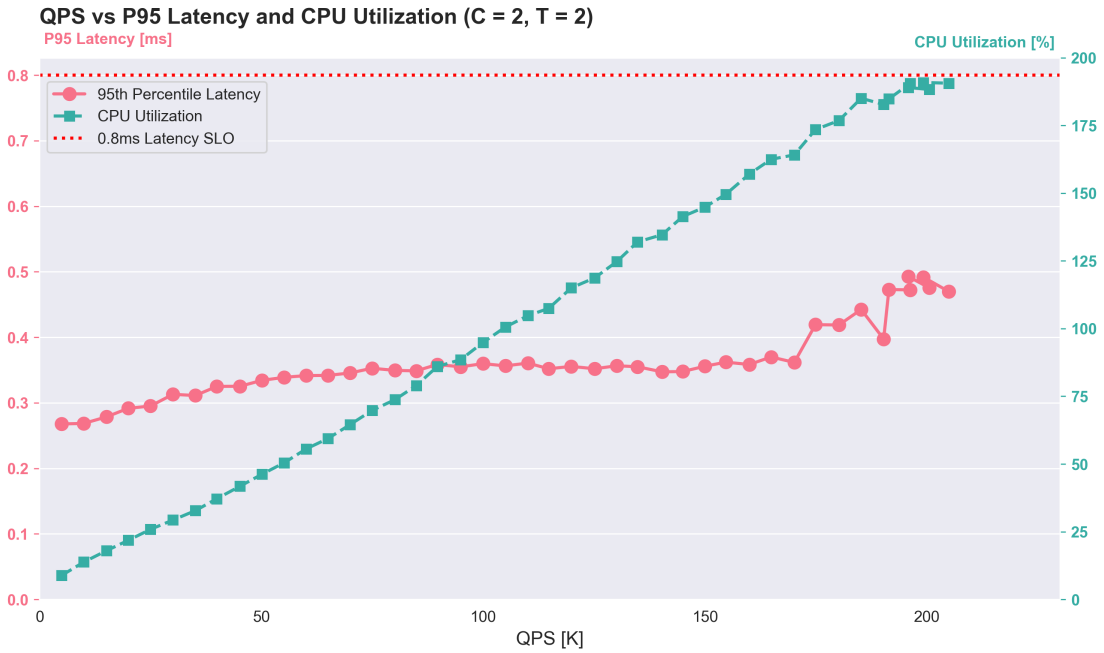

Figure 9: P95 Latency vs CPU Utilization for 2 Threads, 2 Core

2. [**17 points**] You are now given a dynamic load trace for memcached, which varies QPS randomly between 5K and 180K 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 8 -C 8 -D 4 -Q 1000 -c 8 -t 1800 \
           --qps_interval 10 --qps_min 5000 --qps_max 180000
```

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 0.8ms 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 system dynamically adjusts core allocation based on current workload demands. Memcached is allocated between 1-2 cores, with scaling determined by CPU utilization as a proxy for load. When Memcached's core allocation changes, we immediately rebalance our benchmark processes accordingly. The scheduler runs the jobs freqmine and ferret on 3 or 2 cores, which guarantees that they run in an isolated fashion — a desirable property given their sensitivity to interferences. The rest of the jobs are ran alongside each other in 1 or 2 cores and will overlap in 1 core if the load is high, we arranged the ordering such that the jobs likely to overlap with one another don't undermine each other's performance too much (e.g. canneal and radix which are both moderately insensitive to interferences).

- How do you decide how many cores to dynamically assign to memcached? Why?
  Answer: We dynamically assign cores to the memcached service based on the CPU utilization of the process. This serves as a proxy to the QPS target it is currently aiming for, with higher utilization corresponding to higher QPS. In particular, we set a threshold according to the results outlined in Figures 8 and 9. These plots reveal that a good threshold for switching from 1 to 2 cores is when CPU utilization reaches around 80%. To be conservative and prevent as many SLO violations as possible, we set this to 70%. Similarly, when downscaling we choose a threshold in this region (80% this time). The reason for the different thresholds is naturally to prevent getting stuck in an infinite loop. These settings were empirically found to keep a good balance between low SLO

violation and shorter makespan.

- How do you decide how many cores to assign each batch job? Why?
  Answer:
  - blackscholes: this job is run on a single core so that it can be either collocated with other 2-core benchmarks or so that it can run isolated alongside other 1 core jobs. Since blackscholes is relatively robust to interferences, collocating it made sense in this case and helped to reduce the makespan of the jobs.
  - canneal: 1 core minimum, 2 cores if available. The choice to have it run on 2 cores is again to encourage collocation with other processes (2 and 1 core) since the job is also quite robust to interferences. However, we also allow it to downscale to prevent it from interfering with other 1-core benchmarks.
  - dedup: 1 core minimum and maximum because it does not benefit from multi-threading all that much and its runtime is considerably lower. This enables us once again to collocate this with other longer-running benchmarks to reduce the final makespan.
  - ferret: 2 cores minimum, 3 cores if available. Since ferret is one of the longer-running benchmarks and exhibits a favourable parallel speedup, we aim to run it on as many cores as possible while not collocating it with the memcached process. This means we run this job on 3 cores on low load and 2 cores when we are on high load.
  - freqmine: similar to ferret above we run this benchmark on the maximum number of cores available to exploit the parallel speedup and reduce the makespan. This means the job runs on 2 cores under high load and 3 cores if they become available.
  - radix: although radix does benefit from parallelism, our scheduling policy aims to pin it to a single core as it is a relatively shorter benchmark who's runtime can be absorbed into other jobs if successfully collocated. As discussed below, we still aim to achieve some parallelism on this benchmark through multi-threading.
  - vips: we run this benchmark on 2 cores under low load and 1 core minimum under high load, similar to canneal above. This time around the decision to downscale the process is to prevent other jobs from interferring with this benchmark given its sensitivity to collocated processes.

- How many threads do you use for each of the batch job? Why?
  Answer:
  - blackscholes: Although this process showed promising benefits from parallelism and that it is predominantly compute bound, our testing revealed that limiting it to a single thread minimized the interference footprint when ran alongside with canneal.
  - canneal: 2 threads as we noticed we noticed in part 2 that canneal gets a speedup of nearly 2 folds when using 2 threads. 4 threads also showed promising speedup, but given that it is collocated to other jobs, we want to minimize resource contention and leave time for other jobs.
  - dedup: 1 thread, as it produced the poorest speedup in the multi-threaded setting.
  - ferret: 3 threads, we match the number of cores because we know that it is very sensitive to cpu interferences so we try to minimize core contention.
  - freqmine: 3 threads by an analagous reasoning to ferret.

- radix: 4 threads works best because this radix is largely is memory-bound, meaning threads can use CPU time while others stall for memory. This setup fills idle periods during memory accesses, improving overall performance on a single core.
- vips: 2 threads for vips matches its ideal core count, as it shows a 2x speedup on 2 cores and is sensitive to CPU interference. This allocation is also due to its longer execution time.

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

  Answer:
  - Radix and Canneal: both radix and canneal have a reasonably robust interference profile. In particular, canneal is only moderately slowed down by memory bandwidth interference, which given the ample amount of RAM on the machine we are running on shouldn't be a problem in this case.
  - Dedup and Blackscholes: since dedup has zero FLOPs, we expect its interference on the CPU resources to not be as high. Furthermore, given that blackscholes is vulnerable mostly to CPU interferences, it makes sense to pair these two benchmarks together. Furthermore, dedup can be completed in a short time, reducing the duration in which two programs are being collocated.
  - Canneal and Blackscholes: both jobs have a robust interference profile. As in prior collocations with Canneal, memory bandwidth interference shouldn't be a major concern in this setting and thus, the overlap makes sense.
- In which order did you run the batch jobs? Why?

  Order: ferret, canneal, radix, blackscholes, dedup, vips, freqmine

  Why: This order encourages the most two and one core benchmarks to be collocated with one another to reduce the total makespan. In particular, we allow canneal to be collocated with many short-running one-core jobs such as radix and blackscholes. Furthermore, we prioritize running the 3-core isolated jobs whenever the resources are fully available, such as the beginning as we start memcached off on a low load (using a single core).

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

  Answer: The main difference in this setting is that we need to collocate more jobs together in order to achieve a reasonable makespan given that we are much more limited in terms of resources and there is less room for specialization (i.e. assigning jobs to tailored hardware based on profiling experiments). One advantage compared to the previous part, however, is that we have a lot more memory at our disposal, practically eliminating the concern for any meaningful memory bandwidth interference between benchmarks. This latter consideration is particularly impactful in that it allows us to collocate jobs like canneal with other processes without hurting our makespan by much. Another major difference is that, unlike our previous scheduler, we are no longer limited by Kubernetes static resource definitions and thus can dynamically scale the processes to run on different cores as they become available, which in turn helps us minimize idle times a lot more effectively. Finally, having to run the memcached process on the same machine as our benchmark sets an additional challenge as we have to be more conscious about not collocating any jobs with it lest our SLO is violated. The dynamic load adds an extra layer of difficulty to this as we have to intelligently scale memcached up and down while moving

around the benchmarks across cores to guarantee the SLO and maximize resource usage.

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

  Answer: our policy uses a mix of all of the above. In particular, we handle upscaling and downscaling containers based on resource availability using docker cpu-set updates, while we scale the memcached service up or down according to the thresholds described above using taskset updates. In our alternative strategies, we also explored pausing and unpausing containers, particularly those that worked best in 3-core contexts such as ferret but this strategy resulted in some additional overhead and a slightly longer makespan. As a result, we opted out of using it for our final scheduler. The scheduler code is implemented using Python and we used a Strategy design pattern to easily implement and profile different scheduling policies.

- Describe the design choices, ideas and trade-offs you took into account while creating your scheduler (if not already mentioned above). Describe how your policy (and its performance) compares to another policy that you experimented with (in particular to the second-best policy that you designed).

  Answer:

  - SLO-Violation/ Core Availablility: while increasing the threshold makes the cores more available and thus the benchmarks run faster, conversely the slo-violation becomes more likely, these thresholds had to be tuned to find a good middle-ground between both.
  - Reducing Polling Overhead: in order to avoid having to constantly poll the status of our benchmarks, we instead have a job manager that does the bookkeeping of what is running and what is paused. This avoids having to go through the docker service to constantly request the status.
  - Minizing idle cores: We allocated cores to jobs in such a way that we have as many busy cores, this meant that, at times, some jobs that benefited from parallelization were only allocated 1 core, so that it can fit with another job running on 2 cores. This displayed a trade-off between the individual job's runtime and the overall makespan.
  - Avoiding Collocation with Long-Running Jobs: A slowdown on long running jobs affects the makespan more than shorter job, thus we avoid collocating these longer running jobs alongside other jobs and we instead opt to run these jobs in an isolated fashion. This in turn means that the policy has a lower bound on the sequential execution of these two long-running jobs, but we found this setting to be the best empirically.
  - Minimize Pausing Overhead: in our experiments, we noticed that pausing and unpausing containers incurs a small overhead that compounds significantly throughout execution. Thus, we shifted from an initial strategy that worked by pausing containers to our scaling strategy, helping us reduce the makespan by over 10%.
  - Strategy Design Pattern: we implemented a Strategy pattern for flexible scheduler development, allowing us to swap different scheduling algorithms without modifying the underlying infrastructure. This modular approach, paired with the event-driven architecture, enabled quick feedback loops and testing of multiple scheduling strategies with minimal code restructuring.

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 8 -C 8 -D 4 -Q 1000 -c 8 -t 1800 \
           --qps_interval 10 --qps_min 5000 --qps_max 180000 \
           --qps_seed 2333
```

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 > 0.8ms, 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	161.964	4.657
canneal	207.616	0.638
dedup	37.695	9.431
ferret	243.131	3.073
freqmine	253.920	20.786
radix	58.585	3.048
vips	75.982	17.303
total time	766.541	4.724

Answer:

- **Run 1 SLO Violation Ratio**: 2.63%
- **Run 2 SLO Violation Ratio**: 1.33%
- **Run 3 SLO Violation Ratio**: 2.63%

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: The plots for parts A and B for run 1 are presented in Figures 10 and 11 below. Figures 12 and 13 contain the plots for run 2. Finally, figures 14 and 15 contain the plots for the final run.

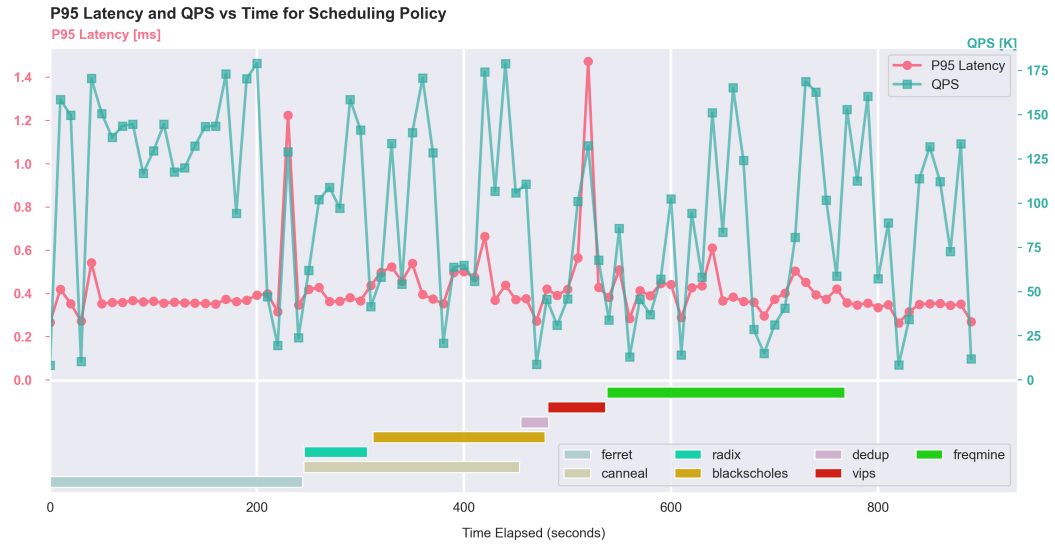

Figure 10: P95 Latency vs QPS Achieved Run 1 (1A)

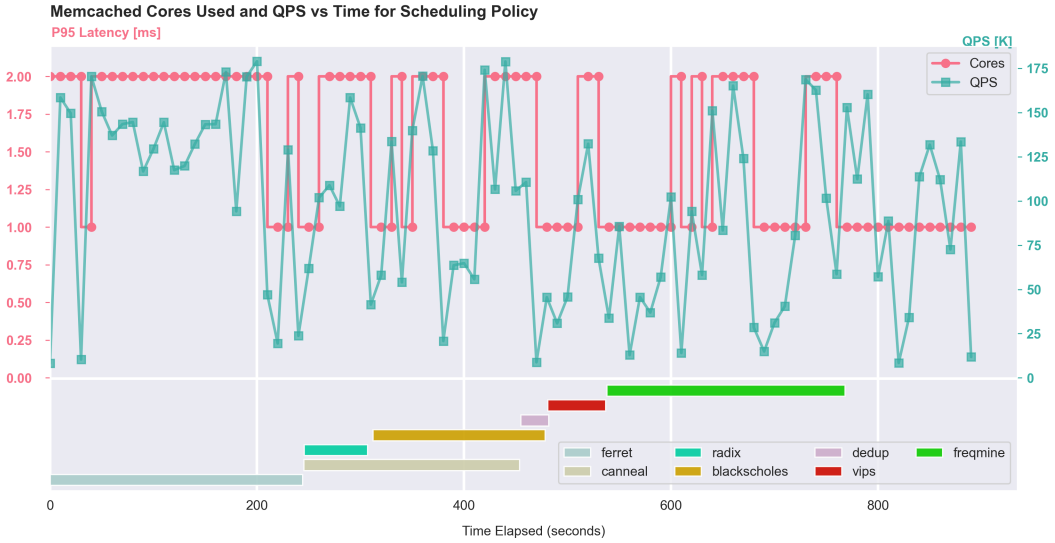

Figure 11: Memcached Cores vs QPS Achieved Run 1 (1B)

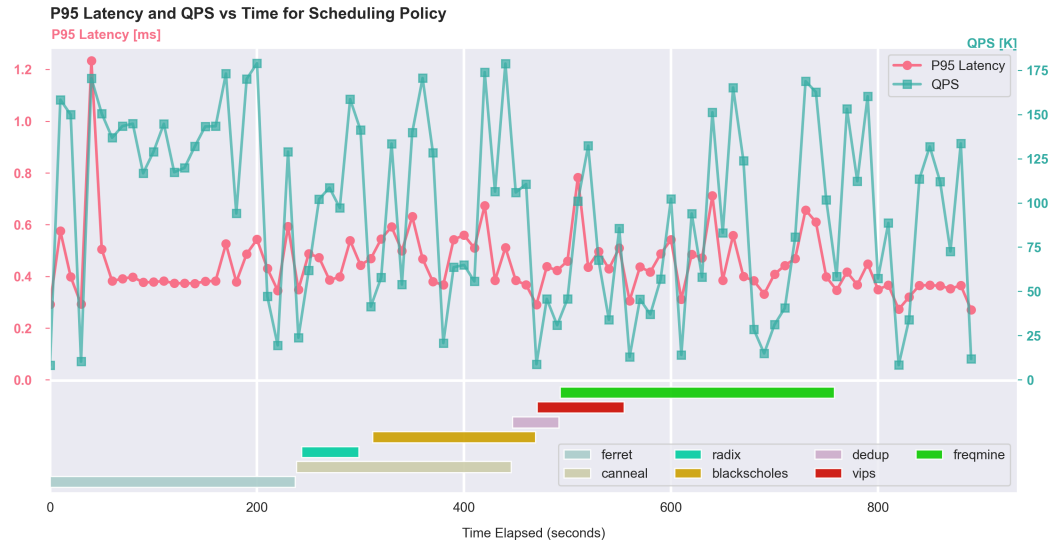

Figure 12: P95 Latency vs QPS Achieved Run 2 (2A)

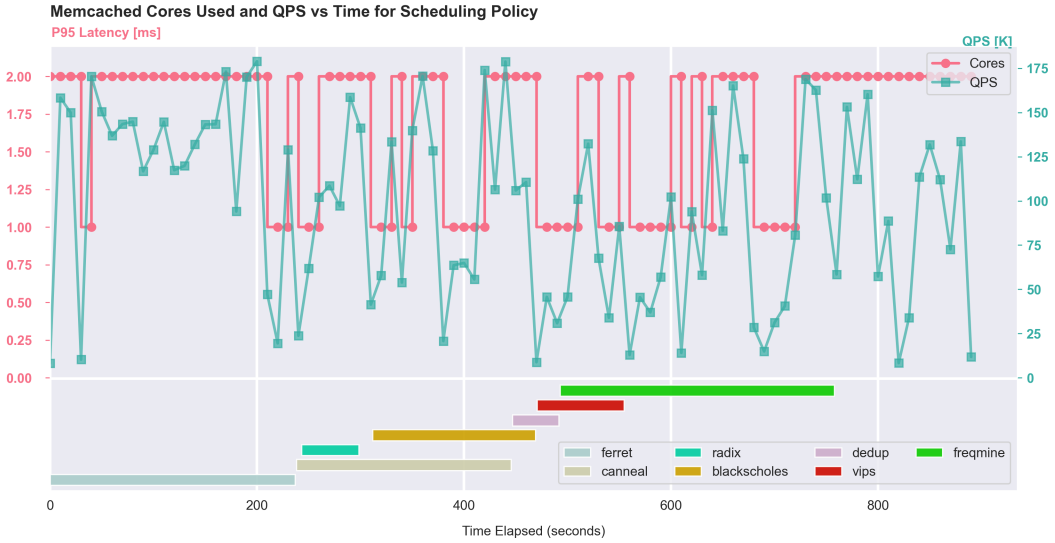

Figure 13: Memcached Cores vs QPS Achieved Run 2 (2B)

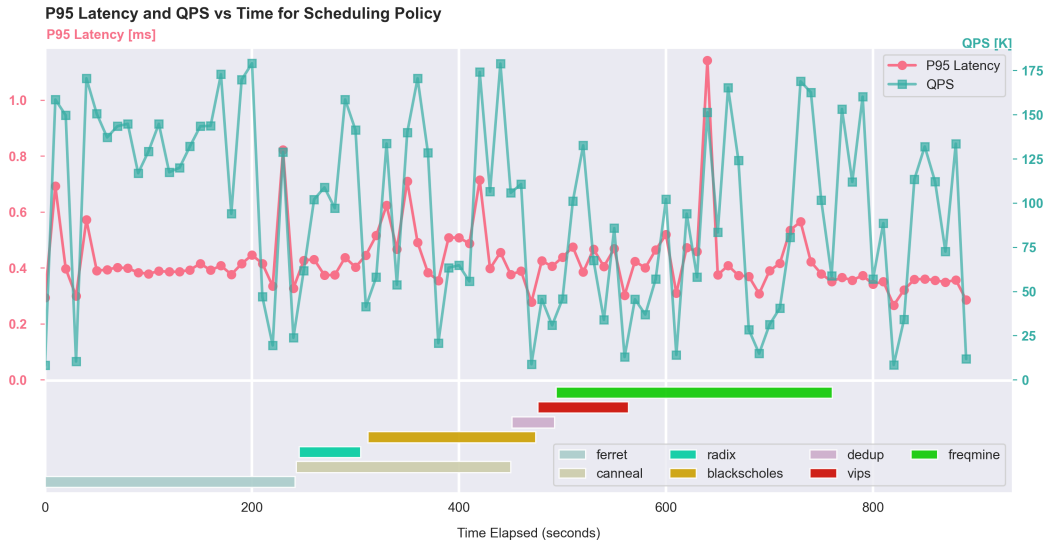

Figure 14: P95 Latency vs QPS Achieved Run 3 (3A)

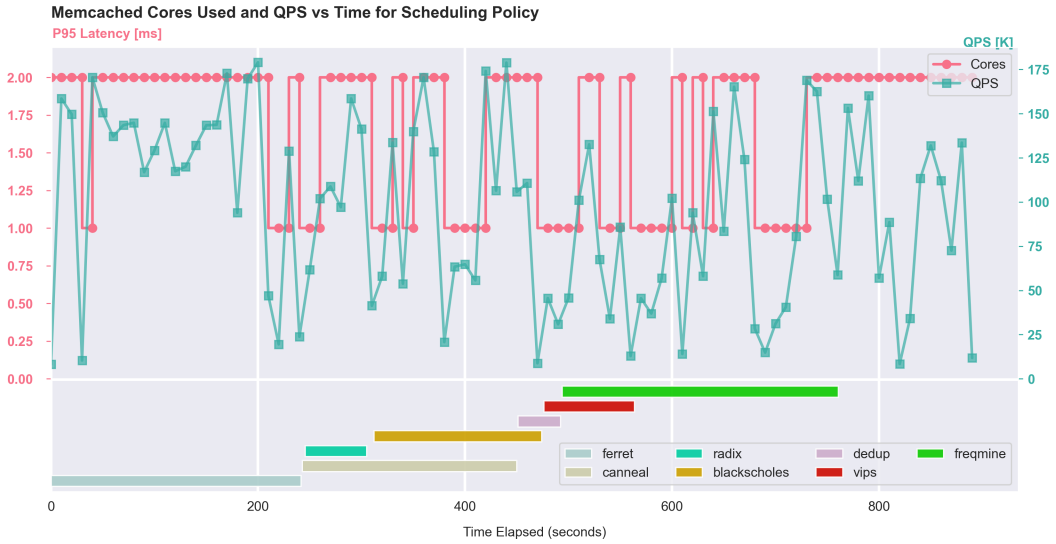

Figure 15: Memcached Cores vs QPS Achieved Run 3 (3B)

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 8 -C 8 -D 4 -Q 1000 -c 8 -t 1800 \
           --qps_interval 5 --qps_min 5000 --qps_max 180000 \
           --qps_seed 2333
```

You do not need to include the plots or table from Question 3 for the 5-second interval. Instead, summarize in 2-3 sentences how your policy performs with the smaller time interval (i.e., higher load variability) compared to the original load trace in Question 3.

Summary: the runtime for the jobs and the final makespan remain relatively unchanged with the increased variability (increase by ~ 0.5 seconds). This is likely due to the fact that our scheduling policy avoids pausing and unpausing any containers in favour of scaling up/down using the taskset command which has a much smaller overhead. The major difference is that by looking at the logs we can see how the change of state from Low to High memcached load happens much more frequently this time, causing containers to scale up and down much more often.

What is the SLO violation ratio for memcached (i.e., the number of datapoints with 95th percentile latency > 0.8ms, 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: our SLO violation ratio in this case is $\frac{27}{180} \approx 15.19\%$.

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 that can be used in our schedule to guarantee the 3% SLO violation ratio is around 9 seconds, so not much lower than the original 10 second interval. The exact value might be slightly lower than this but in order to avoid running out of credits, we decided to stick with this figure in our final plots below.

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

Answer: The main thing that affects the QPS interval given above is the sleep time we encoded into our main controller loop. This ensures that the benchmarks are given sufficient CPU time while still guaranteeing that changes in load are detected relatively fast. Tweaking this value further could improve the performance of the scheduler under more variable loads. However, we saw that empirically the most consistent behaviour could be achieved with a wait time of 1 second for each loop iteration.

Additionally, the fact that our scheduler scales immediately once the thresholds are encountered means that under very variable loads we could end up not allocating sufficient cores to the memcached process, increasing the number of violations. This could be addressed by adding a "cooldown" to the controller which makes it wait for some period of time before changing states. In our experiments however, we found this approach to be very unreliable, resulting in more SLO violations under some loads.

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.

Plots: The plots for parts A and B for run 1 are presented in Figures 16 and 17 below. Figures 18 and 19 contain the plots for run 2. Finally, figures 20 and 21 contain the plots for the final run.

job name	mean time [s]	std [s]
blackscholes	154.295	3.100
canneal	229.908	3.062
dedup	45.832	1.086
ferret	237.193	1.150
freqmine	250.935	16.798
radix	54.618	1.165
vips	71.607	21.114
total time	773.761	1.978

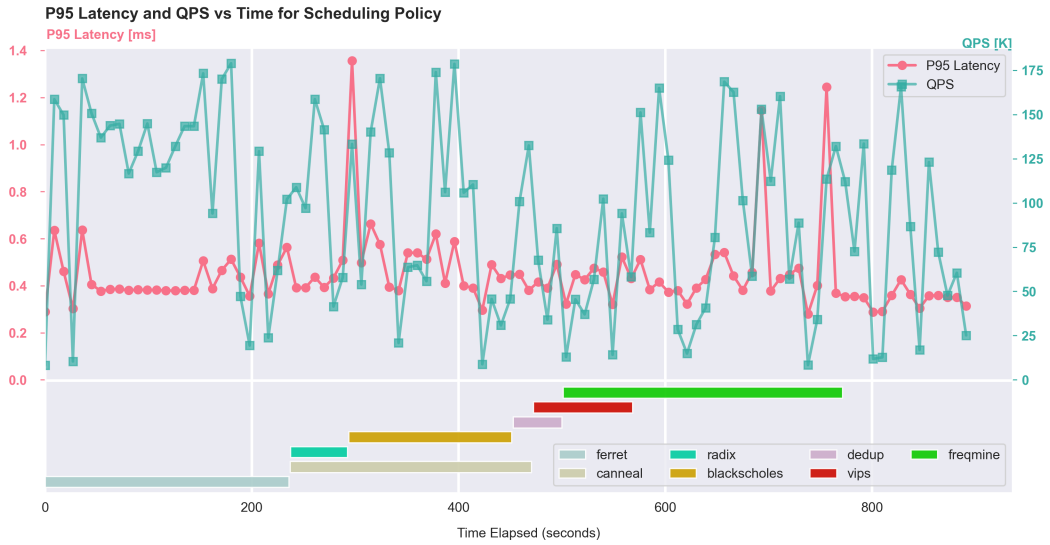

Figure 16: P95 Latency vs QPS Achieved Run 1 (1A)

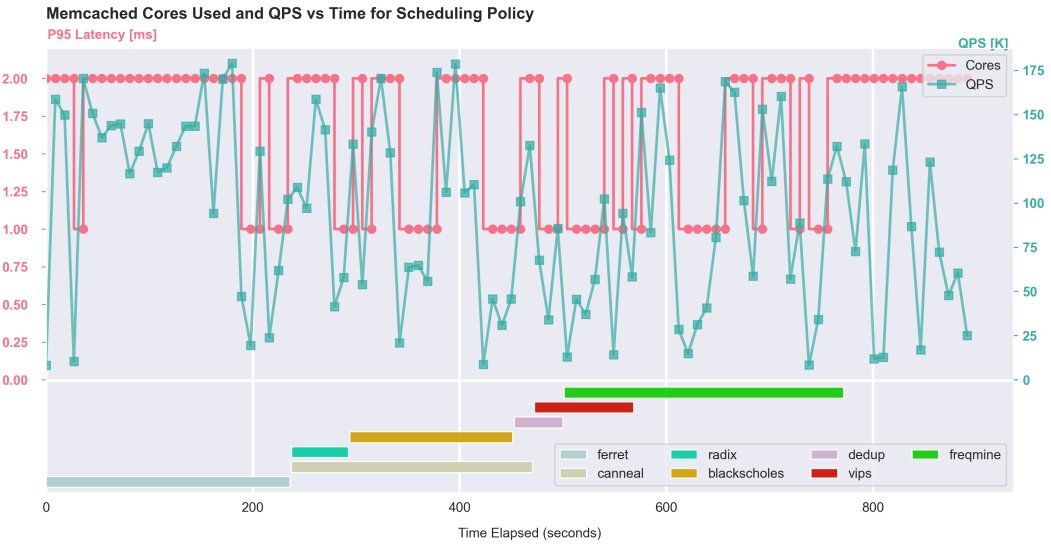

Figure 17: Memcached Cores vs QPS Achieved Run 1 (1B)

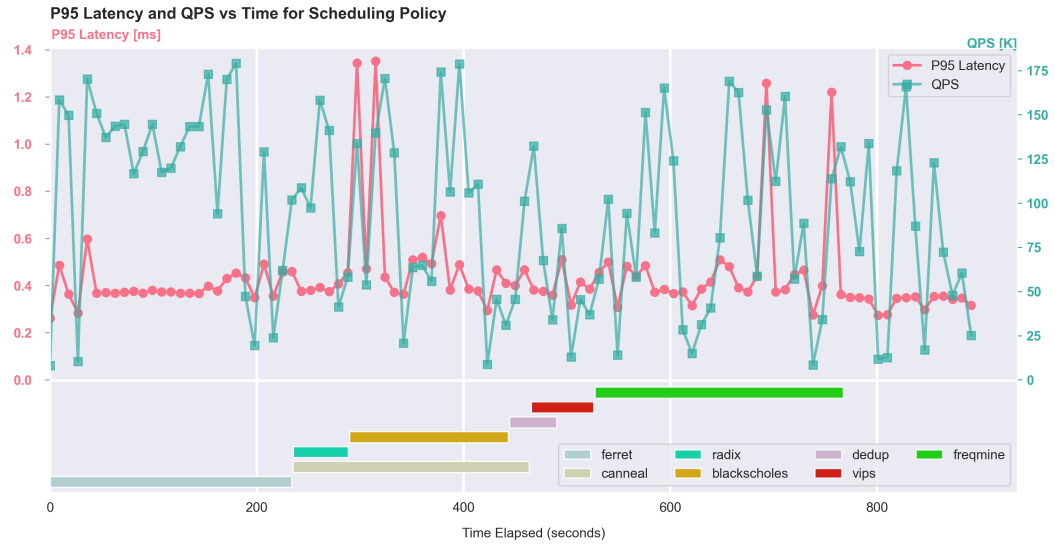

Figure 18: P95 Latency vs QPS Achieved Run 2 (2A)

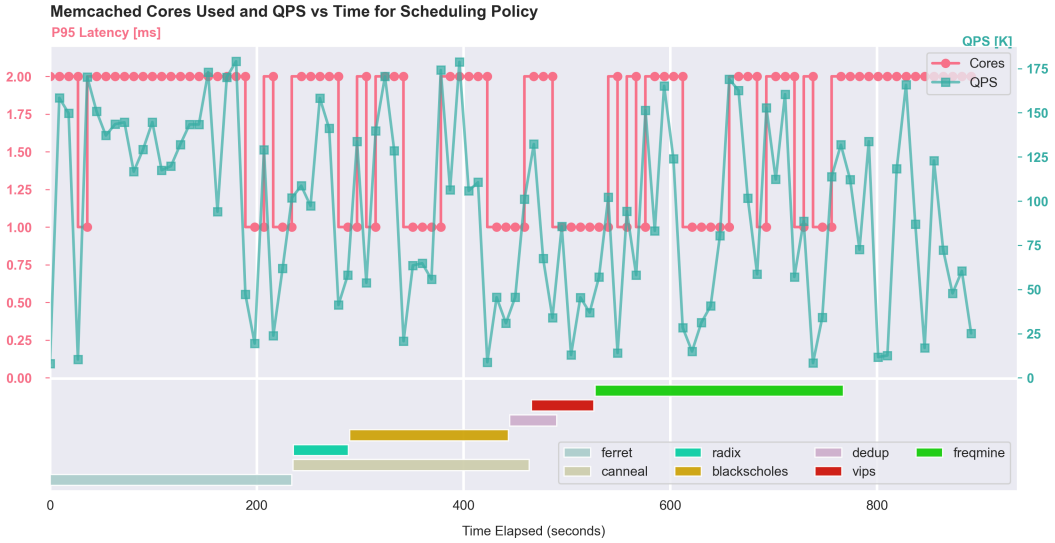

Figure 19: Memcached Cores vs QPS Achieved Run 2 (2B)

Figure 20: P95 Latency vs QPS Achieved Run 3 (3A)

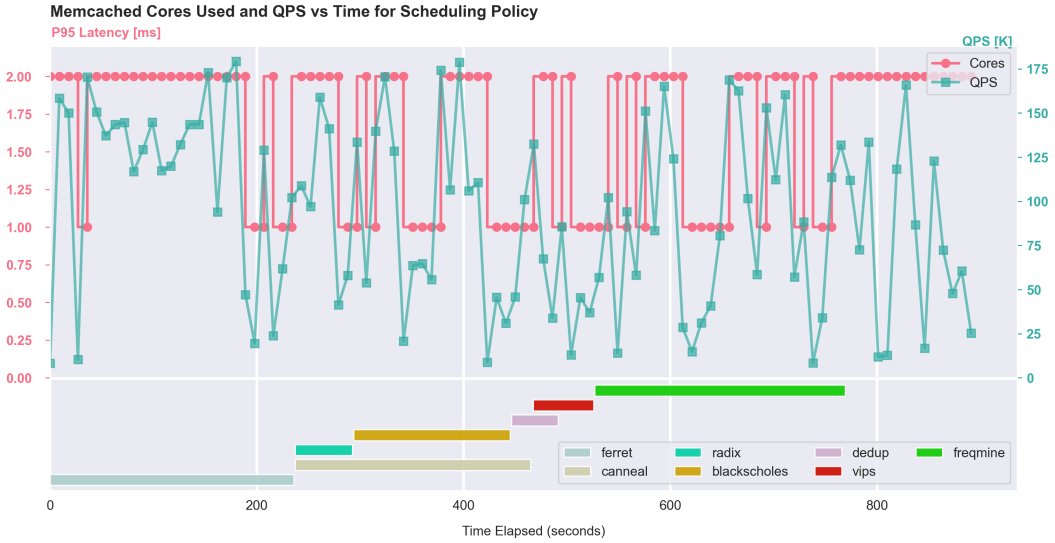

Figure 21: Memcached Cores vs QPS Achieved Run 3 (3B)

<u>Use of AI Tools</u>: Did you use any AI tools for this project? If so, disclose them here and explain what you used the tool(s) for and why. Note that the use of AI tools is NOT needed and NOT expected for the project.

References

Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, page 72–81, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781605582825. doi: 10.1145/1454115.1454128. URL https://doi.org/10.1145/1454115.1454128.