Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Cloud Computing Architecture

Semester project report

Group 008
Riccardo Bollati - 23-943-061
Federica Bruni - 23-947-773
Nicolò Tafta - 21-980-560

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

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.

- Remember to follow the instructions in the project description regarding which files you have to submit.

- Remove this page before generating the final PDF.

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	132.33	3.21
canneal	218.33	10.02
dedup	19.67	9.81
ferret	107.67	2.08
freqmine	192.67	3.06
radix	13.0	1.0
vips	64.0	19.92
total time	197.67	7.64

Answer: Across all three runs, the 95th percentile latency for memcached remains consistent at approximately 0.6ms. Moreover, there are no violations of the SLO in any of the runs, as there are zero instances where the 95th percentile latency exceeds 1ms. Thus the SLO violation ratio is 0.0 for all runs.

Create 3 bar plots (one for each run) of memcached p95 latency (y-axis) over time (x-axis), with annotations showing when each batch job started and ended, also indicating the machine each of them is running on. Using the augmented version of mcperf, you get two additional columns in the output: `ts_start` and `ts_end`. Use them to determine the width of each bar in the bar plot, while the height should represent the p95 latency. Align the x axis so that $x = 0$ coincides with the starting time of the first container. Use the colors proposed in this template (you can find them in `main.tex`). For example, use the vips color to annotate when vips started and stopped, the blackscholes color to annotate when blackscholes started and stopped etc.

Plots:

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

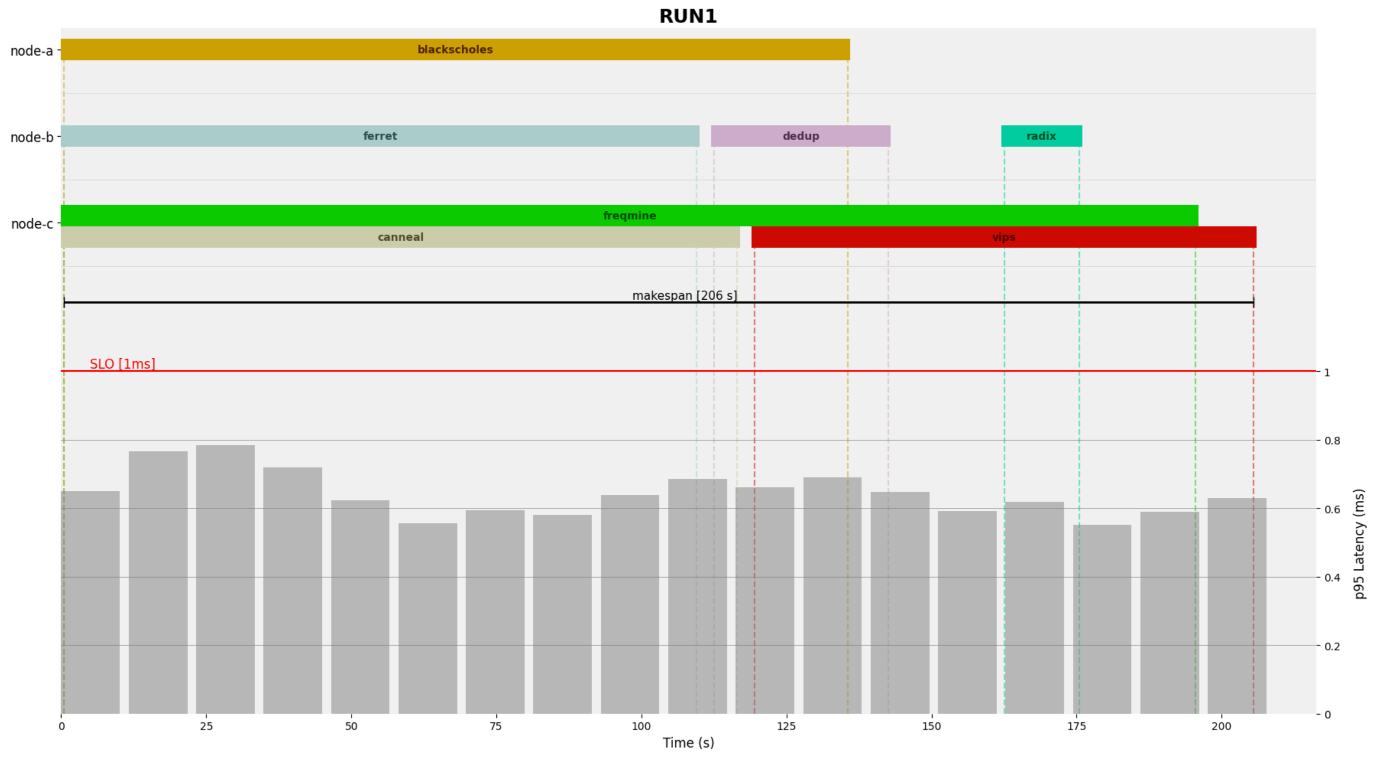

Figure 1: Run 1

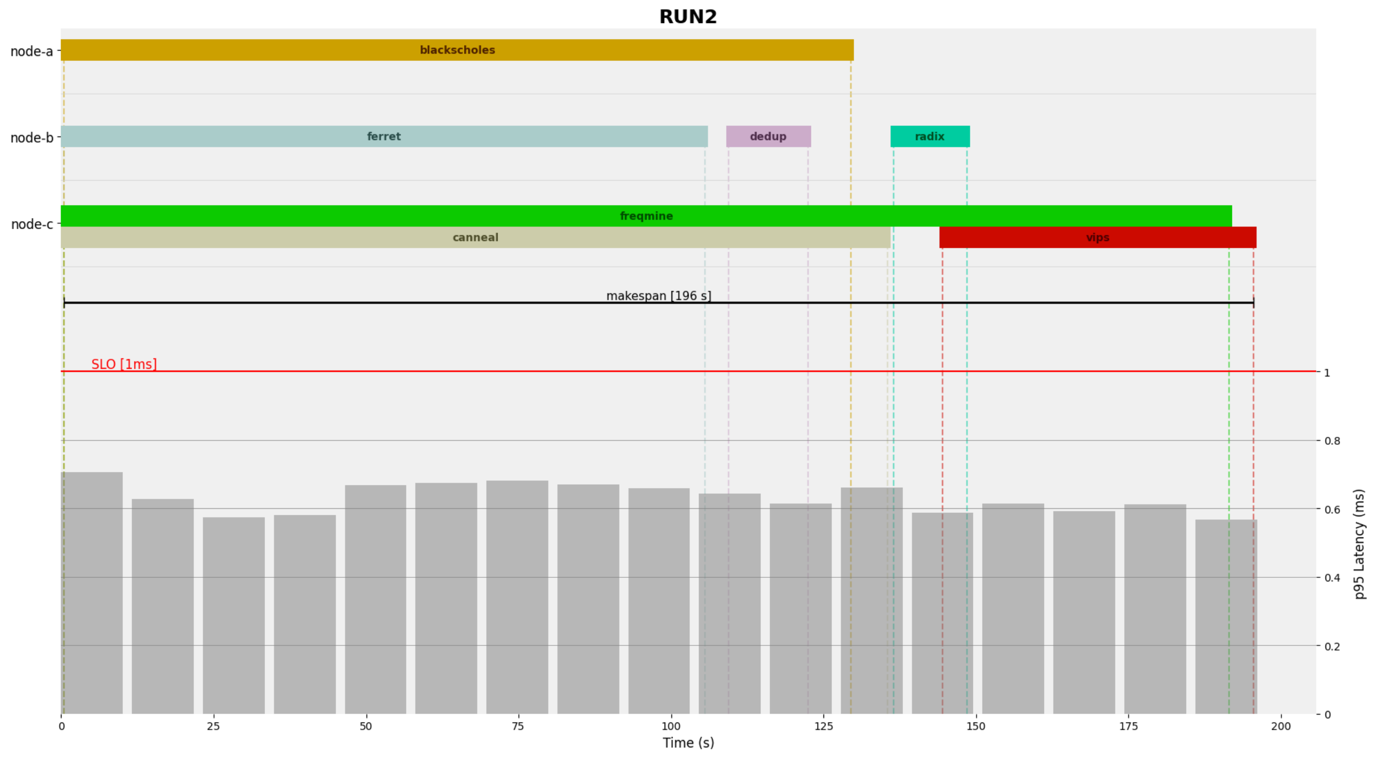

Figure 2: Run 2

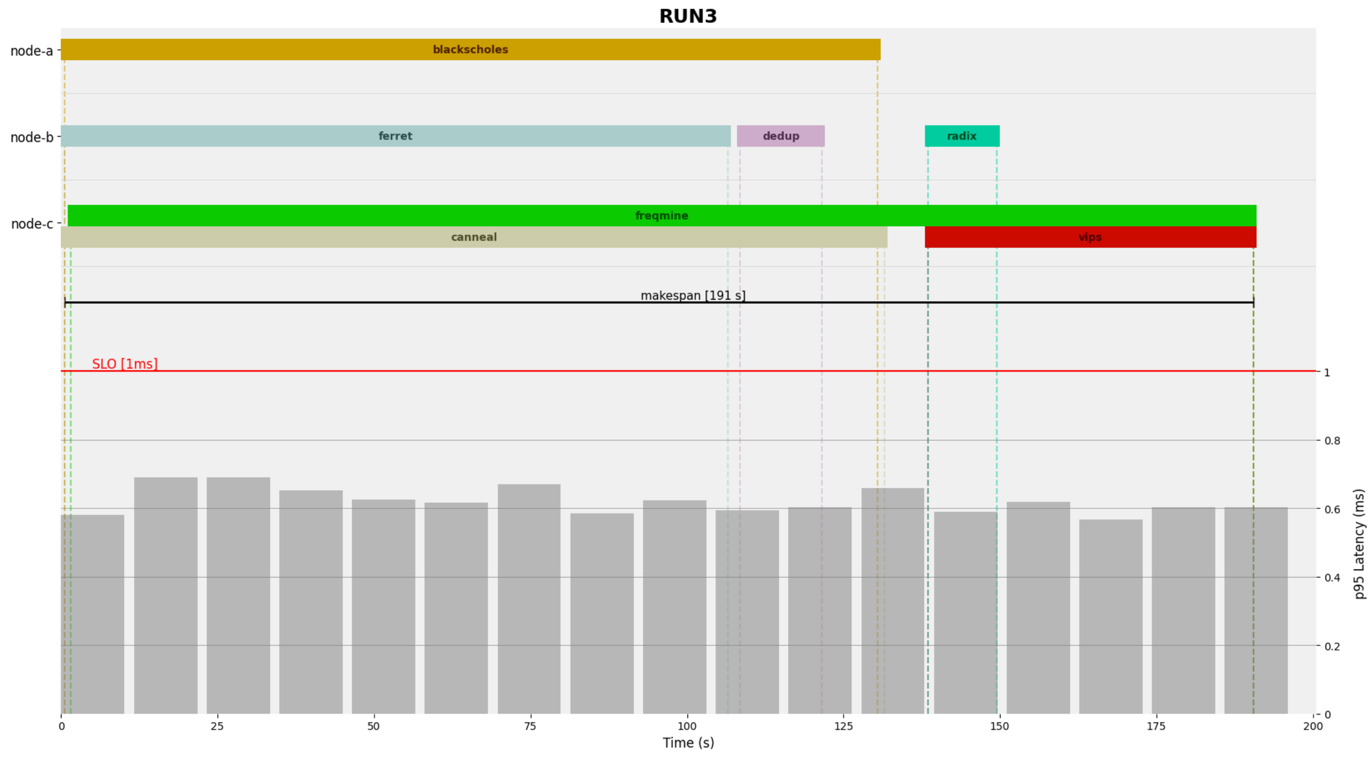

Figure 3: Run 3

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

   - Which node does memcached run on? Why?

     Answer: Memcached runs on node-a because this node features a high-frequency CPU. This aligns well with the computational demands of Memcached tasks, creating an affinity between node-a and Memcached operations.

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

     Answer:

     – blackscholes: This task is executed on Node-a for several reasons. Firstly, it's a lightweight task, perfect for coexisting with memcached on the same machine. Additionally, as discussed in Part 2, it is not worth to be scaled with more than one thread, making it an optimal fit for the remaining core of Node-a.

     – canneal: Canneal operates on Node-c as it requires moderate computational effort. This allows it to run concurrently with freqmine without monopolizing the node's resources for extended periods.

     – dedup: Dedup, being a computationally demanding task, is best suited for Node-b where it can run solo, optimizing performance without competing for resources.

     – ferret: Executed on Node-b, Ferret, the most resource-intensive task, occupies all four cores. This configuration prevents any parallel task allocations, ensuring maximum resource availability.

     – freqmine: Node-c is the chosen venue for freqmine due to its heavy computational and time demands. It uses four cores but does not overwhelm the node, allowing for the simultaneous execution of additional tasks.

  - radix: Radix, being the least demanding both computationally and time-wise, is flexible enough to be run on any node. It is scheduled on Node-b to utilize the availability resulting from the early completion of Ferret.
  - vips: Vips is strategically placed on Node-c, optimally utilizing the available space following canneal and running alongside freqmine.

- Which jobs run concurrently / are colocated? Why?

  Answer:

  - **Colocated jobs**: These are tasks executed on the same node. Ferret, Dedup, and Radix are all colocated on node-b. Similarly, Freqmine, Canneal, and Vips are colocated on node-c.
  - **Concurrent jobs**: These refer to tasks that run simultaneously on the same node. In our scheduling on node-c, both Canneal and Freqmine run concurrently, and likewise, Vips also runs concurrently with Freqmine.

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

  Order: Each node has its scheduling order.

  - **Node-a**: blackscholes
  - **Node-b**: ferret, dedup, radix
  - **Node-c**: freqmine and canneal, vips

  Why: Below are the ordering principles for each node:

  - **Node-a**: Since there is only one job, there is no specific order.
  - **Node-b**: We prioritize allocating the most intensive job first, then the shorter ones.
  - **Node-c**: The order of canneal and vips is arbitrary; they can be interchanged without affecting the final result.

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

  Answer: The allocation of threads for each of the seven batch jobs is determined based on the findings from Part 2.

  - blackscholes: As observed in Part 2, blackscholes demonstrates poor scalability with the number of threads. Therefore, we concluded that using more than one thread for this job is not advantageous. Hence, blackscholes is given 1 thread.
  - canneal: Given that Canneal is among the longest-running jobs, to minimize total execution time, we allocate 4 out of the 8 cores of node-c to it. While the speedup analysis in Part 2 did not reveal significant gains from using additional threads, in this context, it is justified due to Canneal's extended duration. Even a minor reduction in its runtime is crucial, considering it is one of the lengthiest tasks. Thus, we match the number of cores, assigning 4 threads.
  - dedup: Although Dedup doesn't scale efficiently with additional threads, its high resource demands prohibit concurrent execution with other jobs. Consequently, it's allocated all 4 cores exclusively. Even though Part 2 suggested that providing Dedup with more than 2 threads might not be optimal, experimentation with 4 threads proved effective in this context.
  - ferret: Due to its high demands, this job is assigned all four cores of node-b exclusively. Since Ferret demonstrated favorable speedup using four threads in Part 2, we opted to allocate the same number of threads. Thus ferret uses 4 threads.

  - freqmine: The final plot in Part 2 highlighted Freqmine as the job with the best scaling behavior with 4 threads. Considering it also runs on 4 cores of node-c, we opt for 4 threads for freqmine.
  - radix: Part 2 demonstrated that radix scales remarkably well with 8 threads. Given its execution on 4 cores and low-resource requirements, 8 threads are perfect to minimize runtime.
  - vips: With its excellent scalability demonstrated with four threads and its execution on 4 cores, vips is allocated four threads, the optimal choice.

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

  Answer: Regarding Kubernetes, the feature we used is "Resource Management for Pods and Containers." This feature allows us to set a limit on the resources that a job requires and can use once it is launched. By doing so, when we specified the machine for each job, it was sufficient to start all the jobs using Kubernetes, and it managed to launch the jobs as soon as possible by creating a queue. The jobs would then be allocated to the chosen machines as soon as the required resources became available. To avoid issues, we also decided to leave a part of the CPU of the machine free to prevent errors due to CPU availability.

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

  Answer: The scheduler is designed to launch jobs with specified CPU core limits, ensuring each job stays within its allocated resources. Kubernetes then automatically schedules these jobs to fit optimally within the available resources on the nodes. The core limits for each job were set as follows:

  - blackscholes: Requested 0.8 cores and limited to 0.9 cores on node-a.
  - canneal: Limited to 3.8 cores on node-c.
  - dedup: Requested 3.5 cores and limited to 3.5 cores on node-b.
  - ferret: Limited to 3.5 cores on node-b.
  - freqmine: Limited to 3.8 cores on node-c.
  - radix: Limited to 3.5 cores on node-b.
  - vips: Limited to 3.8 cores on node-c.

  The most significant trade-off is the decision to leave a margin free, ie not allocating 100% of the cores. As we increase the margin, a portion of the machines would remain idle, but the execution of the jobs would be less prone to errors.

Please attach your modified/added YAML files, run scripts, experiment outputs and the report as a zip file. You can find more detailed instructions about the submission in the project description file.

Important: The search space of all possible policies is exponential and you do not have enough credits to run all of them. We do not ask you to find the policy that minimizes the total running time, but rather to design a policy that has a reasonable running time, does not violate the SLO, and takes into account the characteristics of the first two parts of the project.

Part 4 [74 points]

1. **[18 points]** Use the following `mcperf` command to vary QPS from 5K to 125K in order to answer the following questions:

```
$ ./mcperf -s INTERNAL_MEMCACHED_IP --loadonly
$ ./mcperf -s INTERNAL_MEMCACHED_IP -a INTERNAL_AGENT_IP  \
          --noload -T 16 -C 4 -D 4 -Q 1000 -c 4 -t 5 \
          --scan 5000:125000:5000
```

a) [7 points] How does memcached performance vary with the number of threads (T) and number of cores (C) allocated to the job? In a single graph, plot the 95th percentile latency (y-axis) vs. achieved QPS (x-axis) of memcached (running alone, with no other jobs collocated on the server) for the following configurations (one line each):

- Memcached with T=1 thread, C=1 core
- Memcached with T=1 thread, C=2 cores
- Memcached with T=2 threads, C=1 core
- Memcached with T=2 threads, C=2 cores

Label the axes in your plot. State how many runs you averaged across (we recommend three runs) and include error bars. The readability of your plot will be part of your grade.

Plots:

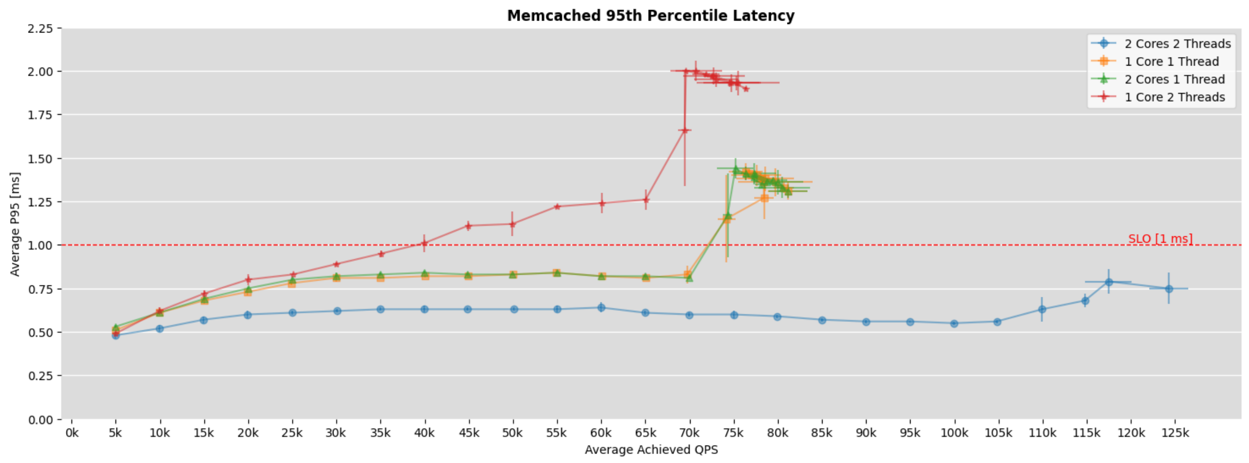

Figure 4: 95th percentile latency vs. QPS for memcached avaraged over 3 runs

We averaged across three runs. Error bars are computed on both the x-axis and y-axis of the graph, calculated as the standard deviation. However, due to some measurements being highly consistent, the error is too minimal to be visible in certain parts of the graph. Notably, the errors become more apparent when the configuration reaches its maximum QPS tolerance, which is expected. Additionally, it's noteworthy that the configuration with 1 core and 2 threads exhibits the largest variance in 95th percentile latency across multiple runs, as evidenced by the error bars.

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:
Figure 4 indicates that memcached performance is least efficient with 2 threads and 1 core, likely due to the overhead of managing an extra thread on a single core and the threads competing with each other. Maximum performance is observed with a configuration of 2 cores and 2 threads, allowing the threads to operate in parallel. This setup is the only one capable of achieving 125K QPS and adhering to the SLO, while other configurations, like 2 cores with 1 thread, yield similar performance to a 1 core 1 thread setup.

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: From the plot it is clearly visible that the only configuration that did not violate the SLO at 125K QPS was 2 Cores 2 Threads. Therefore, we will need 2 Cores and 2 Threads to support the highest load in the trace without violating the 1ms latency 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: In case it is possible to adjust the number of cores allocated to memcached dynamically within a QPS range of 5K to 125K, while keeping the number of threads fixed when launching the memcached job, the best choice is to use 2 memcached threads (T) to maintain a 1ms 95th percentile latency SLO throughout the load fluctuations. This is because the configuration of 2 cores and 2 threads is the only one proven capable of managing 125K QPS. By fixing the number of threads to 2, we ensure that in case of high QPS we can dynamically schedule memcached on 2 cores and thus adopt the safe configuration that never violates the SLO.

d) [8 points] Run memcached with the number of threads T that you proposed in (c) and measure performance with $C = 1$ and $C = 2$. Use the aforementioned `mcperf` command to sweep QPS from 5K to 125K.

Measure the CPU utilization on the memcached server at each 5-second load time step.

Plot the performance of memcached using 1-core ($C = 1$) and using 2 cores ($C = 2$) in **two separate graphs**, for $C = 1$ and $C = 2$, respectively. In each graph, plot achieved QPS on the x-axis, ranging from 0 to 130K. In each graph, use two y-axes. Plot the 95th percentile latency on the left y-axis. Draw a dotted horizontal line at the 1ms latency SLO. Plot the CPU utilization (ranging from 0% to 100% for $C = 1$ or 200% for $C = 2$) on the right y-axis. For simplicity, we do not require error bars for these plots.

Plots:

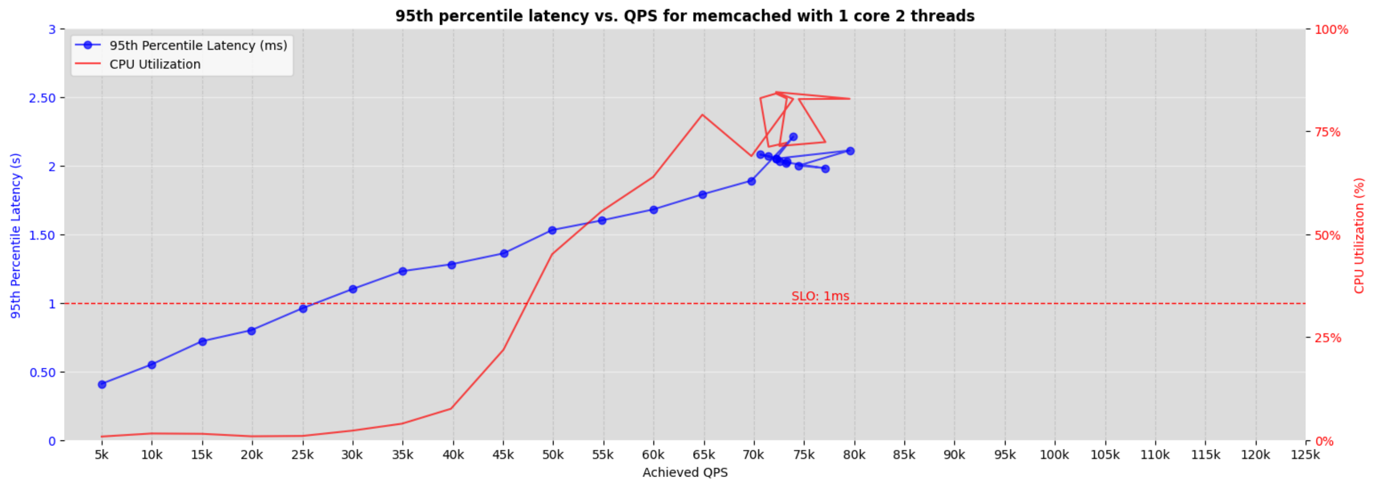

Figure 5: Plot 3A.

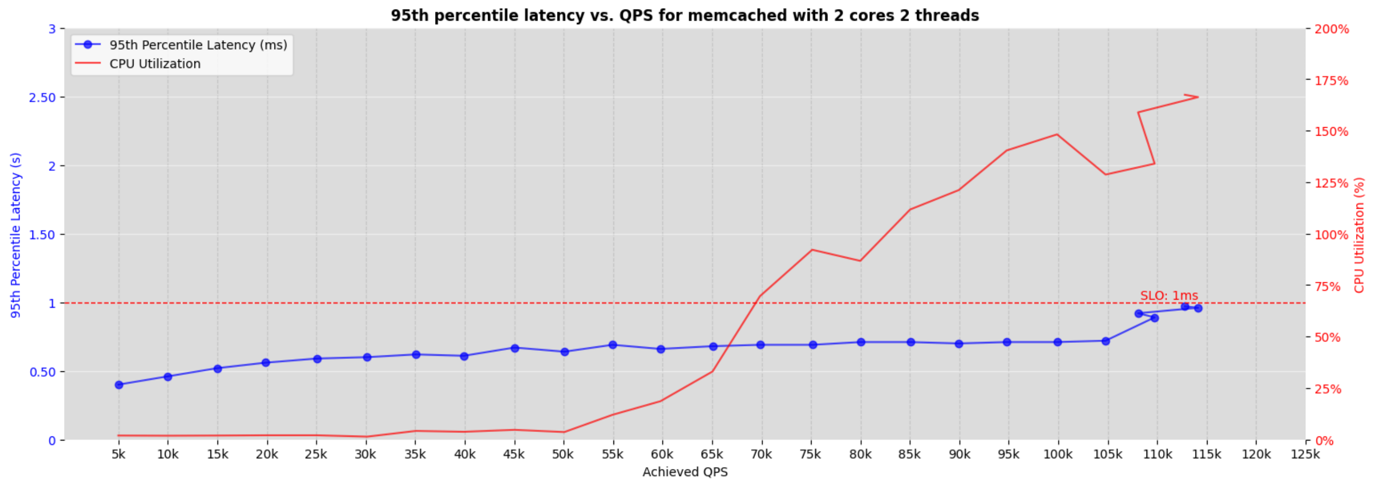

Figure 6: Plot 3B.

2. [**17 points**] You are now given a dynamic load trace for memcached, which varies QPS randomly between 5K and 100K in 10 second time intervals. Use the following command to run this trace:

```
$ ./mcperf -s INTERNAL_MEMCACHED_IP --loadonly
$ ./mcperf -s INTERNAL_MEMCACHED_IP -a INTERNAL_AGENT_IP \
          --noload -T 16 -C 4 -D 4 -Q 1000 -c 4 -t 1800 \
          --qps_interval 10 --qps_min 5000 --qps_max 100000
```

Note that you can also specify a random seed in this command using the `--qps_seed` flag.

For this and the next questions, feel free to reduce the mcperf measurement duration (`-t` parameter, now fixed to 30 minutes) as long as you have at the end at least 1 minute of memcached running alone.

Design and implement a controller to schedule memcached and the benchmarks (batch jobs) on the 4-core VM. The goal of your scheduling policy is to successfully complete all batch jobs as soon as possible without violating the 1ms 95th percentile latency for memcached. **Your controller should not assume prior knowledge of the dynamic load trace. You should design your policy to work well regardless of the random seed.** The batch jobs need to use the native dataset, i.e., provide the option `-i native` when running them. Also make sure to check that all the batch jobs complete successfully and do not crash. Note that batch jobs may fail if given insufficient resources.

Describe how you designed and implemented your scheduling policy. Include the source code of your controller in the zip file you submit.

- Brief overview of the scheduling policy (max 10 lines):

  Answer: The scheduling policy manages CPU resources between memcached and PARSEC benchmark jobs using Docker containers, with two job queues handling single-core and two-core jobs. Memcached primarily runs on core 0, occasionally using core 1 based on load. The policy monitors CPU usage of cores used by memcached and adjusts its CPU affinity as needed. When CPU usage is high, the job on core 1 is paused, reallocating its CPU to memcached. When usage is low, the paused job resumes or a new job starts, confining memcached to core 0. Core 1 is assigned to dedup, vips, and radix, while cores 2 and 3 are allocated exclusively to ferret and freqmine in sequence. Then blackscholes and canneal are run concurrently, each on one core. After these jobs complete, jobs from the single-core queue can use cores 2 and 3.

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

  Answer: To ensure a stable start and mitigate initial spikes observed in previous runs, memcached initially utilizes two cores. Subsequently, we track the sum of CPU utilization of cores 0 and 1 every 0.05 seconds for 40 iterations, totaling 2 seconds of monitoring. If the maximum observed CPU utilization remains under 30%, the policy assigns memcached to a single core. Conversely, for scaling up to two cores, we directly examine the instant CPU usage of core 0. If it exceeds 30%, memcached is scaled back up to use both cores. This approach facilitates a smooth transition to two cores, guarding against latency spikes, as it requires only a single instance of high CPU usage to trigger the transition. However, the policy is more cautious when scaling down to one core, as it considers CPU utilization over the past 2 seconds, ensuring stability by disregarding short-lived CPU spikes.

- How do you decide how many cores to assign each batch job? Why?
  Answer:
  - blackscholes: This job is assigned to a single core. According to the analysis in Part 1-2, blackscholes is not resource-intensive and does not require more than one core.
  - canneal: Similar to blackscholes, this job is also assigned to one core.
  - dedup: Dedup is also assigned to a single core, specifically core 1, which is subject to pauses when memcached's load surpasses the threshold. This is because dedup is one of the fastest jobs, so pausing it does not significantly penalize the total makespan.
  - ferret: Ferret is assigned to cores 2 and 3 exclusively, according to the policy. As a parallel search engine, ferret is a CPU-intensive task that benefits from multiple cores to improve its parallel execution.
  - freqmine: Similar to ferret, freqmine also utilizes cores 2 and 3. As an application which performs frequent itemset mining, freqmine is a computationally intensive task and thus requires multiple cores to optimize its performance.
  - radix: Radix is assigned to core 1 after Dedup completes. Once the jobs in the two-core queue finish, Radix is moved to cores 2 and 3. The reasoning is similar to Dedup: Radix is a fast job, so pausing it does not significantly impact the total makespan. Additionally, when Radix is moved to two cores, it finishes running almost immediately.
  - vips: As an image processing system, vips requires high computational resources, but according to the analysis in Part 1-2, it is less intensive than ferret or freqmine and is therefore allocated only one core. However, since the jobs in the two-core queue finish before Radix starts running, vips is dynamically moved to cores 2 and 3 after Radix completes.

- How many threads do you use for each of the batch jobs? Why?
  Answer:
  - blackscholes: For blackscholes, we allocate only one thread because, as shown in Part 1-2, this job does not scale well with more than one thread. Additionally, since it runs on a single core, multiple threads would not significantly improve performance and could increase overhead due to context switching.
  - canneal: With canneal assigned to a single core, providing it with more threads would mainly introduce overhead without significant performance improvement.
  - dedup: Similar to canneal and blackscholes, dedup receives only one thread, given its allocation on a single core.
  - ferret: Utilizing cores 2 and 3, Ferret employs two threads to fully utilize the parallel processing capacity of these cores, resulting in significant performance enhancement. Also the final plot in Part 2 indicates that ferret is one of the best jobs at scaling with two threads.
  - freqmine: Like Ferret, freqmine uses two threads corresponding to its allocation on cores 2 and 3, optimizing its performance by leveraging the computational potential of these cores. Its speedup analysis reported similar behaviour to ferret which is a confirm of using two threads for this job.

  - radix: Radix is allocated 8 threads as most of the times it ends up running on two cores and its efficient parallelization makes it suitable for this configuration. This choice optimizes performance while utilizing the available computational resources effectively.
  - vips: With vips showing good scalability with four threads and often running on two cores, utilizing 4 threads maximizes performance without introducing unnecessary overhead.

  The optimal thread configuration was determined based on earlier analyses in parts 1 and 2 of the assignment, where the ideal thread count was identified for each job. Subsequent experimentation required only minor adjustments.

- Which jobs run concurrently / are collocated and on which cores? Why?
  Answer:
  - **Colocated jobs**: These are tasks executed on the same core. However, in our scheduling, we did not allocate jobs to the same core, hence there are no colocated jobs.
  - **Concurrent jobs**: These are tasks that run simultaneously. In our scheduling, dedup runs concurrently with all jobs on cores 2-3 until completion, meaning it runs concurrently with: ferret, freqmine, canneal, and blackscholes. Additionally, canneal and blackscholes are scheduled to run concurrently on cores 2 and 3, respectively.

  Memcached initially operates on core 0 and can utilize core 1 if CPU usage surpasses ??. On core 1, the jobs dedup, vips, and radix run sequentially in the order specified. However, they may be temporarily halted if memcached requires core 1 due to high CPU usage. Jobs ferret and freqmine exclusively occupy cores 2 and 3, running sequentially on these cores. After ferret and freqmine complete, blackscholes and canneal are allocated to run on cores 2 and 3, respectively, concurrently. Once these jobs finish, other jobs initially assigned to core 1 can also utilize cores 2 and 3. This design aims to prevent resource conflicts, ensuring each job has dedicated access to a core. Memcached, being critical, is granted priority access, allowing it to claim an additional core if necessary.

- In which order did you run the batch jobs? Why?
  Order:
  - **Core 1**: dedup, radix, vips.
  - **Core 2 and 3**: ferret, freqmine, at the same time: blackscholes on core2 and canneal on core 3.

  Why:
  - **Core 1**: The order of tasks on this core is arbitrary and can be altered without impacting the final result. Instead, jobs are allocated to cores in a sequential manner to avoid resource contention and ensure optimal execution.
  - **Core 2 and 3**: We prioritize allocating the most resource-intensive tasks first, ferret and freqmine. Subsequently, we allocate the less demanding but time-consuming tasks: canneal and blackscholes.
- How does your policy differ from the policy in Part 3? Why?
  Answer: In Part 3, the policy did not involve dynamically adjusting the number of running jobs based on the CPU utilization of the system and neither changing the number

of cores of memcached or other applications. In contrast, the policy in Part 4 involves dynamically adjusting the number of running containers based on the CPU utilization of the system. The policy uses the psutil library to monitor the CPU utilization of the system, and pauses and unpauses containers based on the CPU utilization. If the CPU utilization is too high, some of the containers are paused to free up resources. If the CPU utilization is low, some of the paused containers are resumed to make use of the available resources. The main reason for this difference is that the policy in Part 4 is more flexible and can adapt to changes in the CPU utilization of the system. This policy, as the SLO violations of 0.0% report, is robust to a dynamic load trace to memcached. Using policy of Part 3 with that dynamic load trace would result in much poorer results, especially in terms of SLO violations. By dynamically adjusting the number of running containers, the policy can ensure that the system is using the available resources efficiently, while also maximizing the performance of the system.

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

  Answer: The implementation of the policy includes setting the CPU affinity for the 'memcached' process via the taskset command, as well as configuring, initiating, and managing Docker containers designated for PARSEC jobs on specific CPU cores. This process involves dynamically pausing and resuming Docker containers, along with updating their assigned CPU cores as necessary. Additionally, the policy requires monitoring CPU usage and adjusting the CPU affinity for 'memcached' based on the observed CPU load. The function we have used to handle the jobs execution and respective containers are the followings:

  - **client.containers.run()**: this function is used to start the containers, inside the function we have used the argument: *cpuset_cpus* to select in which cores to run the container.
  - **current_1.pause()** & **current_1.unpause()**: these tho function allowed us to pause and unpause the contaner runnning in the core 1 when the cpu utilization become too hight due to memcached activity.
  - **subprocess.run(f"sudo taskset -a -cp 0-1 {MEM_ID}", shell=True, capture_output=False, stdout = subprocess.DEVNULL)** : this is the command we have used to let memcached use also the core 1 when needed and also to switch back to just the core 0 when the activity decreased.
  - **current_4.update(cpuset_cpus="2-3")** : this is the function we have used to re-allocate the containers to the desired cores.

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

  Answer: While most of these aspects were already explained before,let's delve a bit deeper into the trade-offs involved in job allocation. Based on the insights from Part 1-2, ferret and freqmine, being the most CPU-intensive, were given priority allocation, each exclusively assigned two cores in sequence. For jobs designated to run on core 1 and be paused when necessary for memcached, we opted for faster tasks to minimize overall makespan. As a result, dedup, radix, and vips were selected for this queue due to their efficient execution. Finally, considering the less intensive yet time-consuming nature of blackscholes and canneal, we decided to run them concurrently on core 2 and core 3, respectively, to minimize the total makespan while effectively utilizing available resources.

3. [**23 points**] Run the following `mcperf` memcached dynamic load trace:

```
$ ./mcperf -s INTERNAL_MEMCACHED_IP --loadonly
$ ./mcperf -s INTERNAL_MEMCACHED_IP -a INTERNAL_AGENT_IP \
         --noload -T 16 -C 4 -D 4 -Q 1000 -c 4 -t 1800 \
         --qps_interval 10 --qps_min 5000 --qps_max 100000 \
         --qps_seed 3274
```

Measure memcached and batch job performance when using your scheduling policy to launch workloads and dynamically adjust container resource allocations. Run this workflow 3 separate times. For each run, measure the execution time of each batch job, as well as the latency outputs of memcached. For each batch application, compute the mean and standard deviation of the execution time [2] across three runs. Compute the mean and standard deviation of the total time to complete all jobs - the makespan of all jobs. Fill in the table below. Also, compute the SLO violation ratio for memcached for each of the three runs; the number of data points with 95th percentile latency > 1ms, as a fraction of the total number of datapoints. The SLO violation ratio should be calculated during the time from when the first batch-job-container starts running to when the last batch-job-container stops running.

job name	mean time [s]	std [s]
blackscholes	100.67	2.52
canneal	267.67	4.04
dedup	607.67	99.47
ferret	187.33	2.89
freqmine	258.0	1.0
radix	123.67	88.79
vips	59.00	1.73
total time	790.67	10.6

Answer: Across all three runs, the 95th percentile latency for memcached remains consistent at approximately 0.7ms. Moreover, there are no violations of the SLO in any of the runs, as there are zero instances where the 95th percentile latency exceeds 1ms. Thus the SLO violation ratio is 0.0 for all runs.

- **Run 1**: 0.0% SLO violation (0 data points over 100)
- **Run 2**: 0.0% SLO violation (0 data points over 100)
- **Run 3**: 0.0% SLO violation (0 data points over 100)

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

[2] 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:

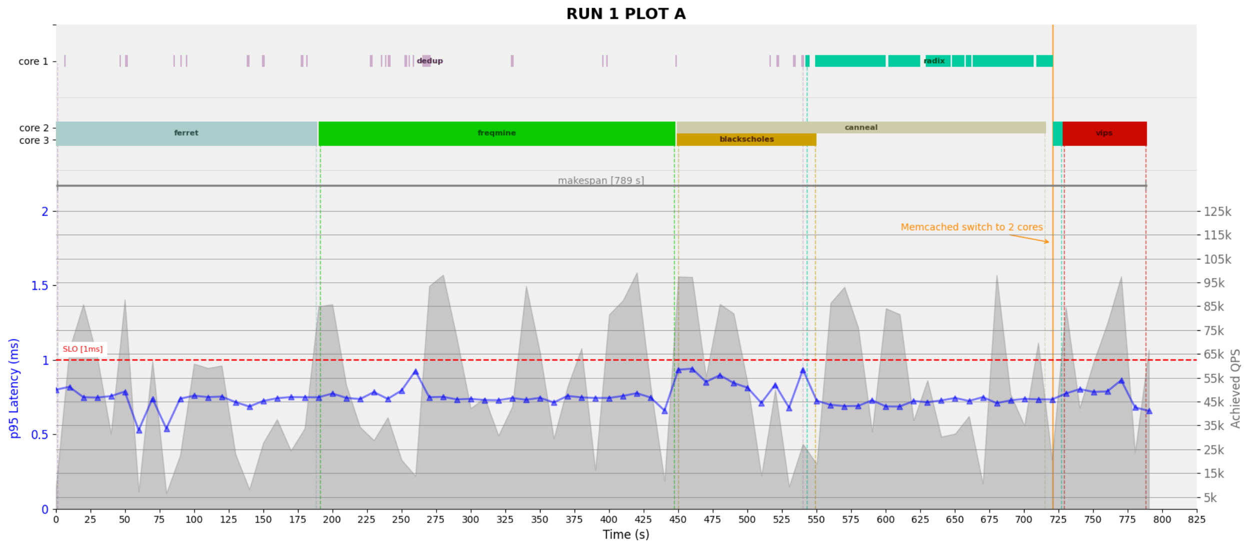

Figure 7: Plot 1A.

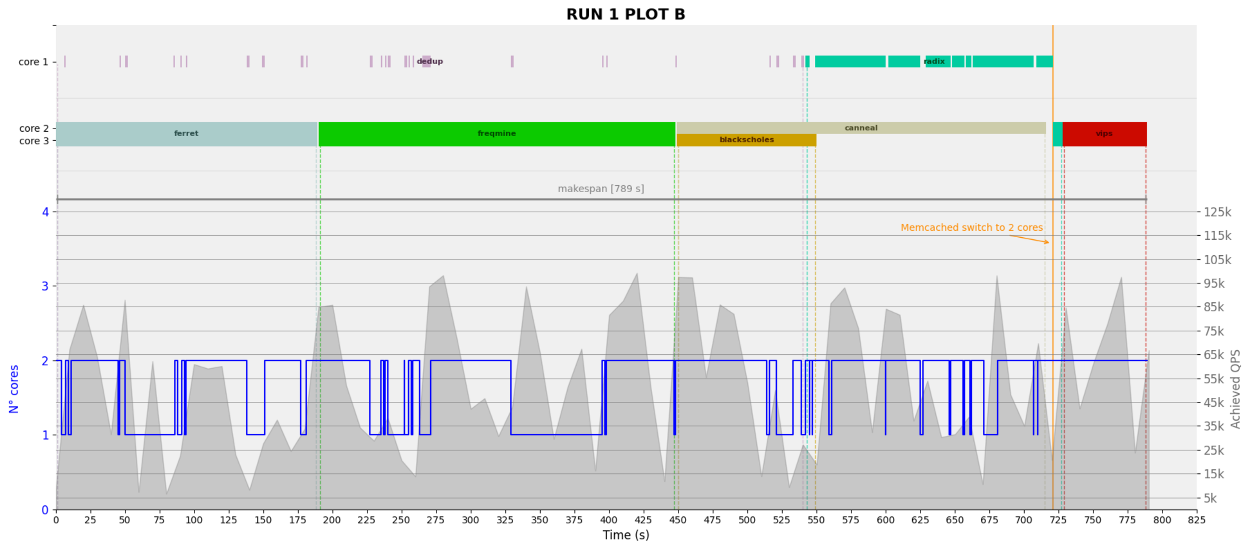

Figure 8: Plot 1B.

.

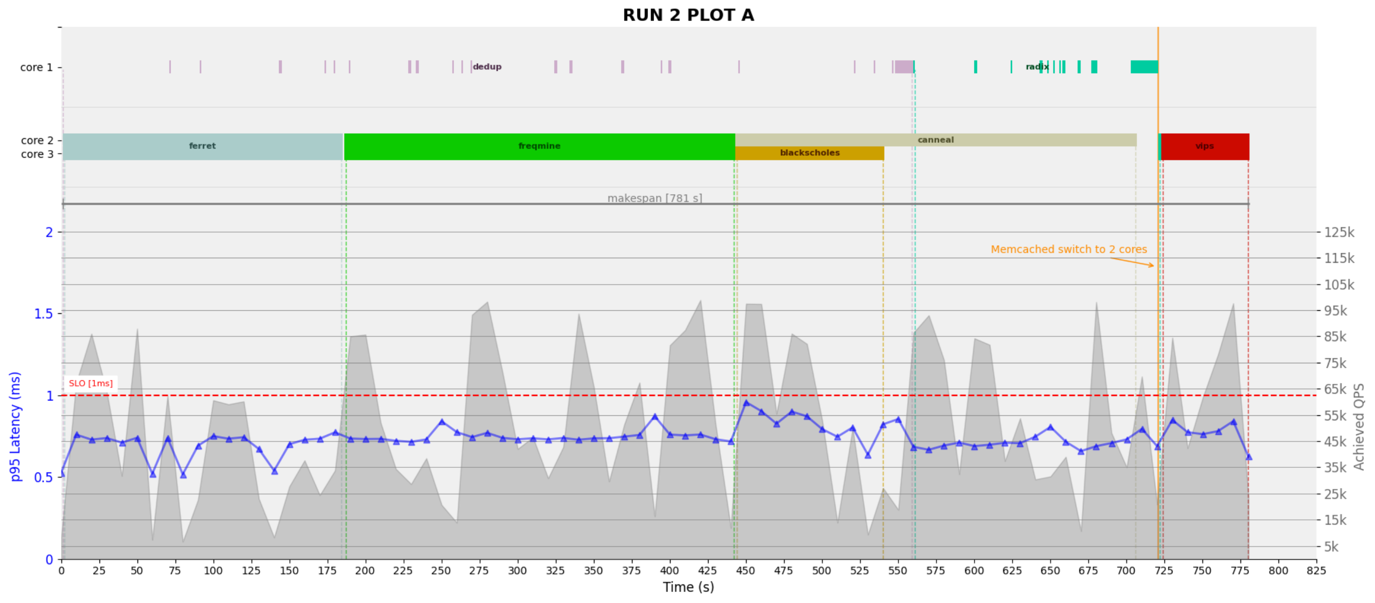

Figure 9: Plot 2A.

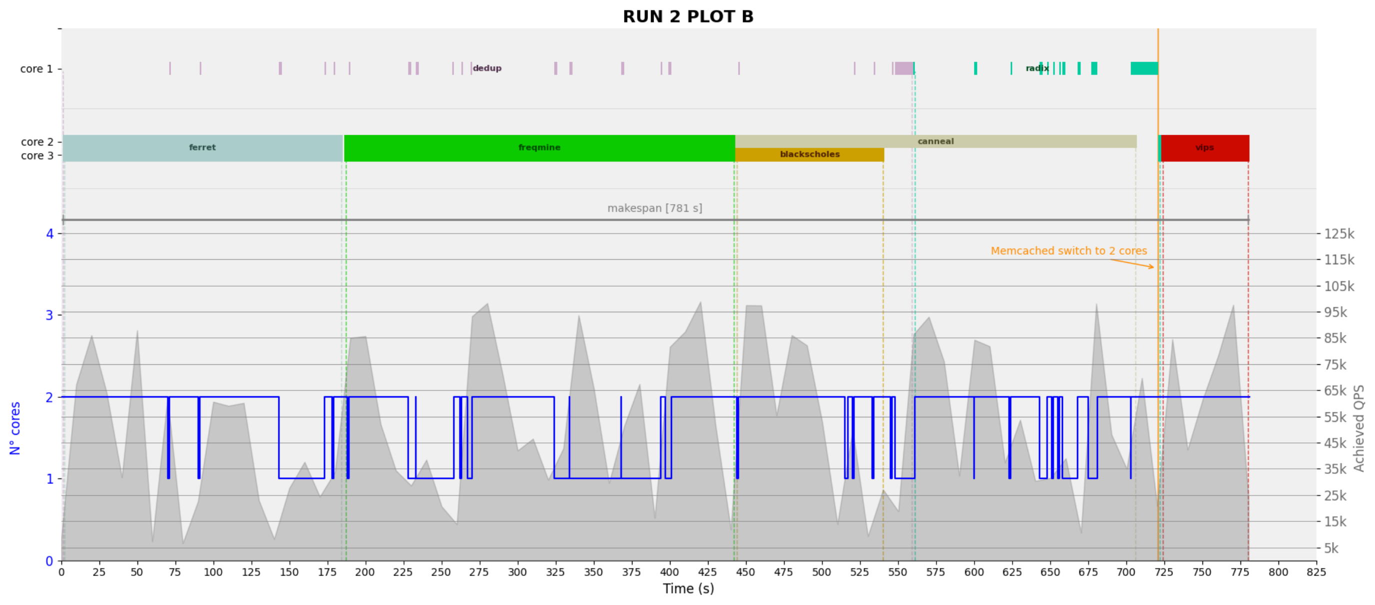

Figure 10: Plot 2B.

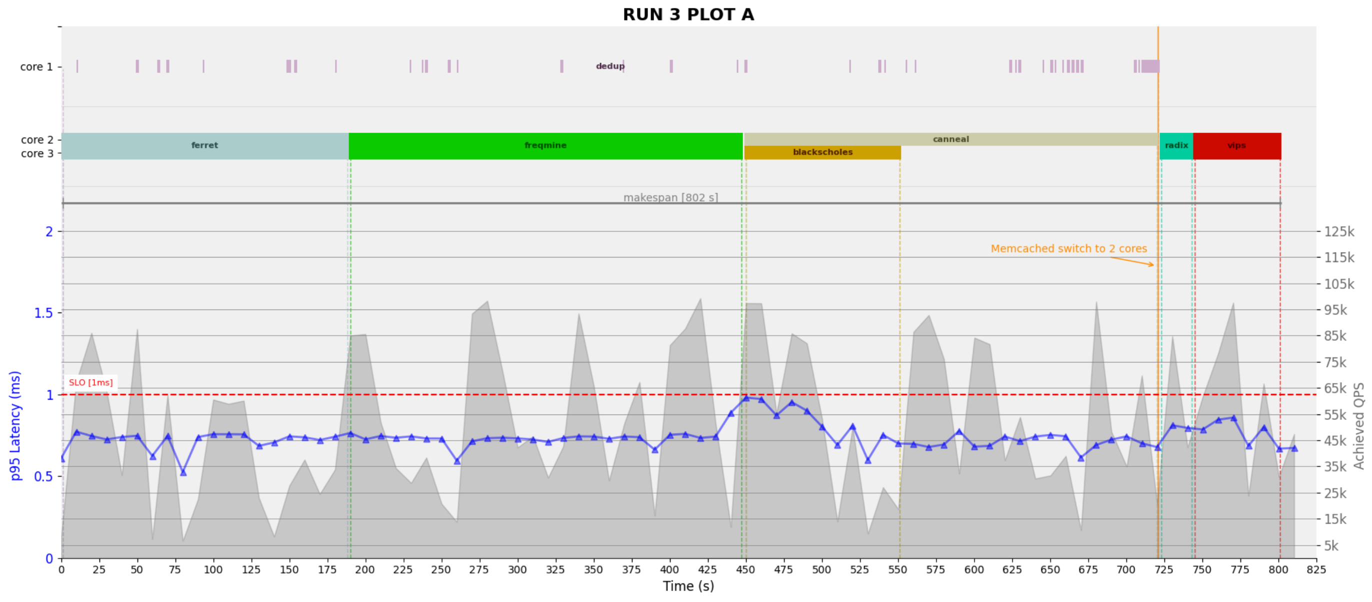

Figure 11: Plot 3A.

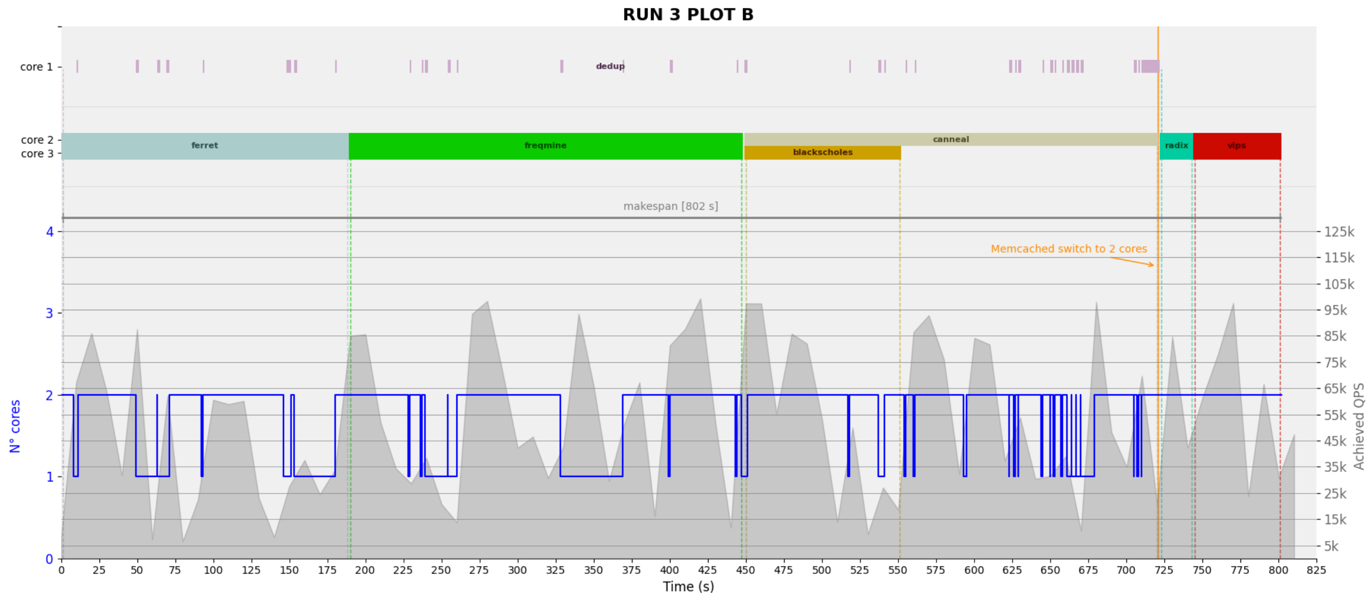

Figure 12: Plot 3B.

4. [**16 points**] Repeat Part 4 Question 3 with a modified `mcperf` dynamic load trace with a 5 second time interval (`qps_interval`) instead of 10 second time interval. Use the following command:

```
$ ./mcperf -s INTERNAL_MEMCACHED_IP --loadonly
$ ./mcperf -s INTERNAL_MEMCACHED_IP -a INTERNAL_AGENT_IP \
          --noload -T 16 -C 4 -D 4 -Q 1000 -c 4 -t 1800 \
          --qps_interval 5 --qps_min 5000 --qps_max 100000 \
          --qps_seed 3274
```

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

Summary: We averaged over three runs and surprisingly the policy performed better on average with a 5-second time interval compared to the 10-second interval over three runs, possibly because the 2-second decision-making period aligns better with this shorter interval. The average total makespan decreased to 771.0s compared to 790.67s with the 10-second interval, but the standard deviation increased to 33.1 s from 10.6 s, indicating improved average performance but reduced stability.

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:

- **Run 1**: 0.0% SLO violation (0 data points over 220)
- **Run 2**: 0.0% SLO violation (0 data points over 220)
- **Run 3**: 0.0% SLO violation (0 data points over 450)

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 scheduler we've developed places a high priority on adhering to the Service Level Objective (SLO). Consequently, even with smaller QPS intervals, it effectively maintains the memcached SLO violation ratio at approximately 0%. Notably, as the QPS changes more rapidly, the scheduler adopts a more vigilant approach. Because of how the scheduler is designed, when the QPS varies more frequently, there is a higher probability that in the history, which the scheduler considers to move memcached to 1 core, there is high CPU utilization. This makes it more difficult for memcached to transition from 2 cores to 1 core, making the policy safer and enabling it to adhere to the SLO.

As the QPS interval decreases, the scheduler becomes increasingly protective, often allocating memcached to remain on two cores. Through our research, we've identified that the 2-second interval marks the threshold where memcached still dynamically transitions between one and two cores. Under this interval memcached is always allocatded two cores. This value is not casual, it represents the window within which we observe and make decisions regarding scaling back to a single core. If the QPS change occur more frequently, such as every second, our scheduler proactively allocates two cores to memcached, prioritizing the overarching goal of adhering to the SLO.

Consequently, our scheduler demonstrates robustness across varying QPS intervals, consistently maintaining SLO violations at approximately 0.0%. For a QPS interval of 2 seconds, the observed SLO violations are as follows:

- **Run 1**: 0.02% SLO violation (8 data points over 403)
- **Run 2**: 0.01% SLO violation (5 data points over 400)
- **Run 3**: 0.02% SLO violation (8 data points over 400)

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

Answer: The interval of 2s represents the window within which we observe and make decisions regarding scaling back to a single core. Thus it marks the threshold where memcached still dynamically transitions between one and two cores. Under this interval memcached is always allocatded two cores.

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

job name	mean time [s]	std [s]
blackscholes	97.67	2.08
canneal	258.33	8.50
dedup	705.67	27.79
ferret	201.0	0.0
freqmine	253.33	1.15
radix	36.0	26.0
vips	56.0	0.0
total time	799.67	2.89

Plots:

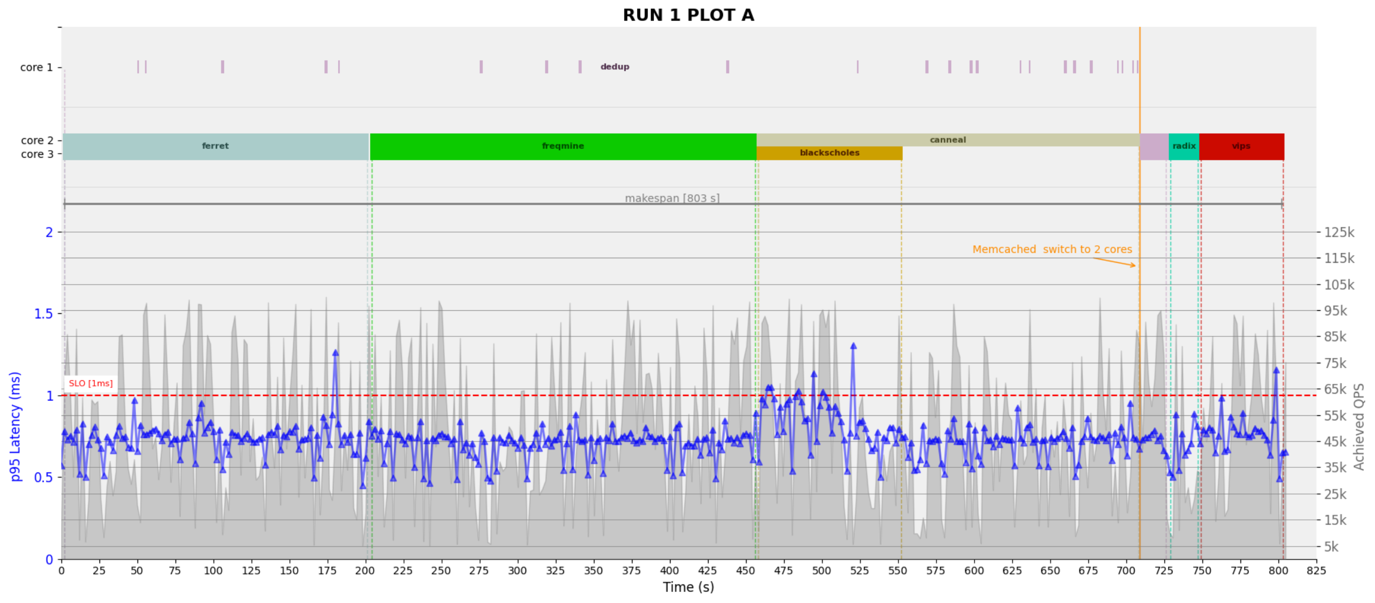

Figure 13: Plot 1A.

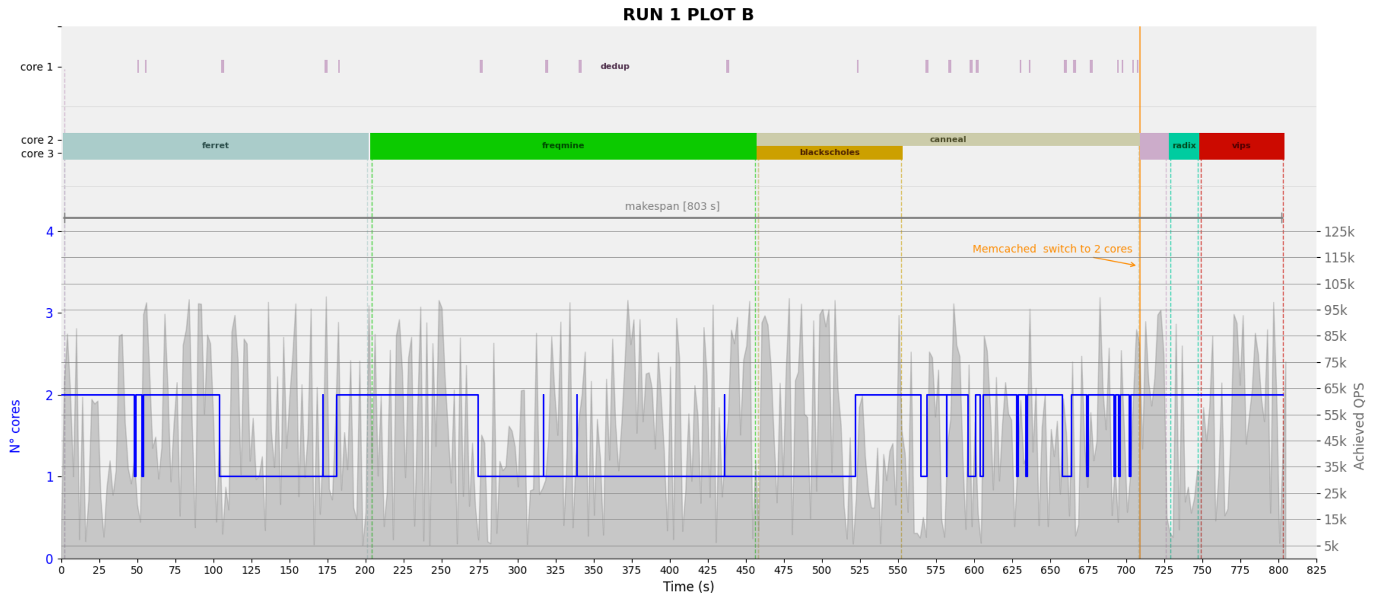

Figure 14: Plot 1B.

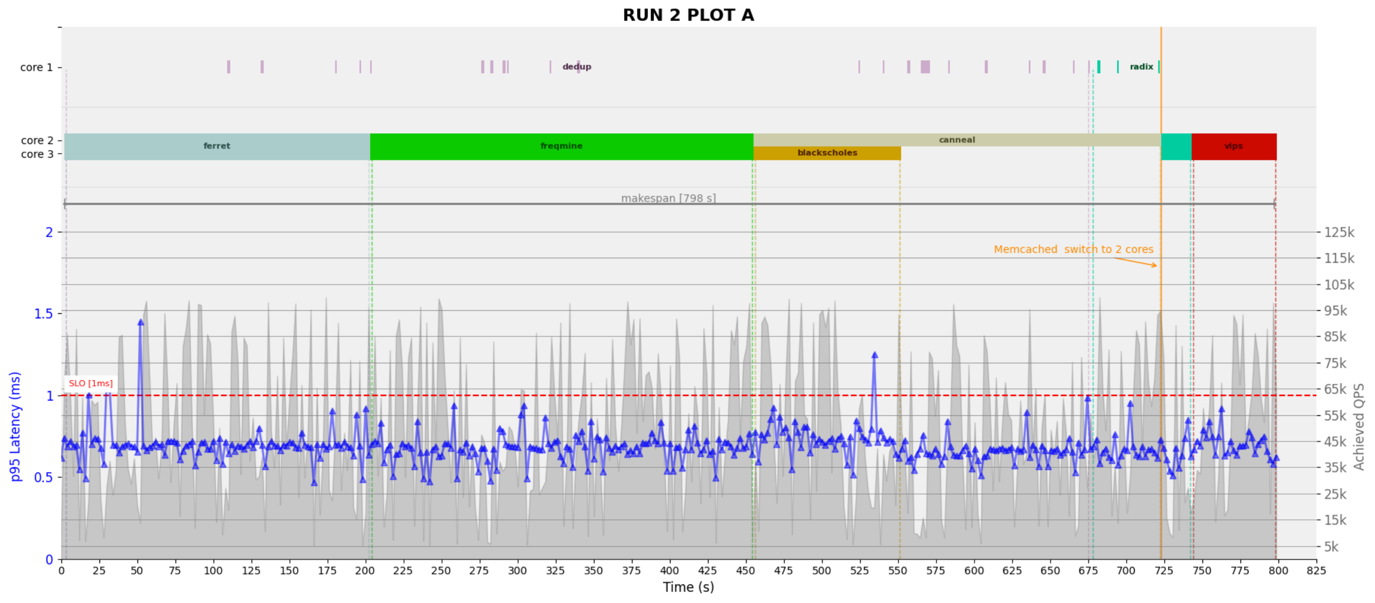

Figure 15: Plot 2A.

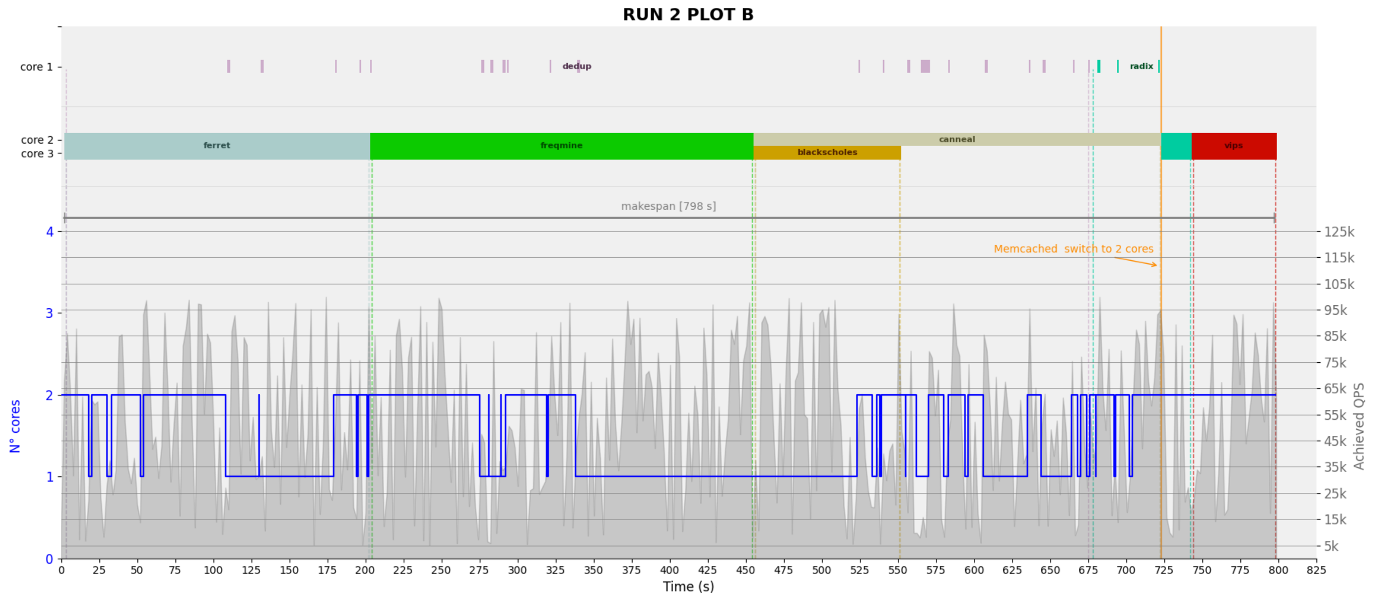

Figure 16: Plot 2B.

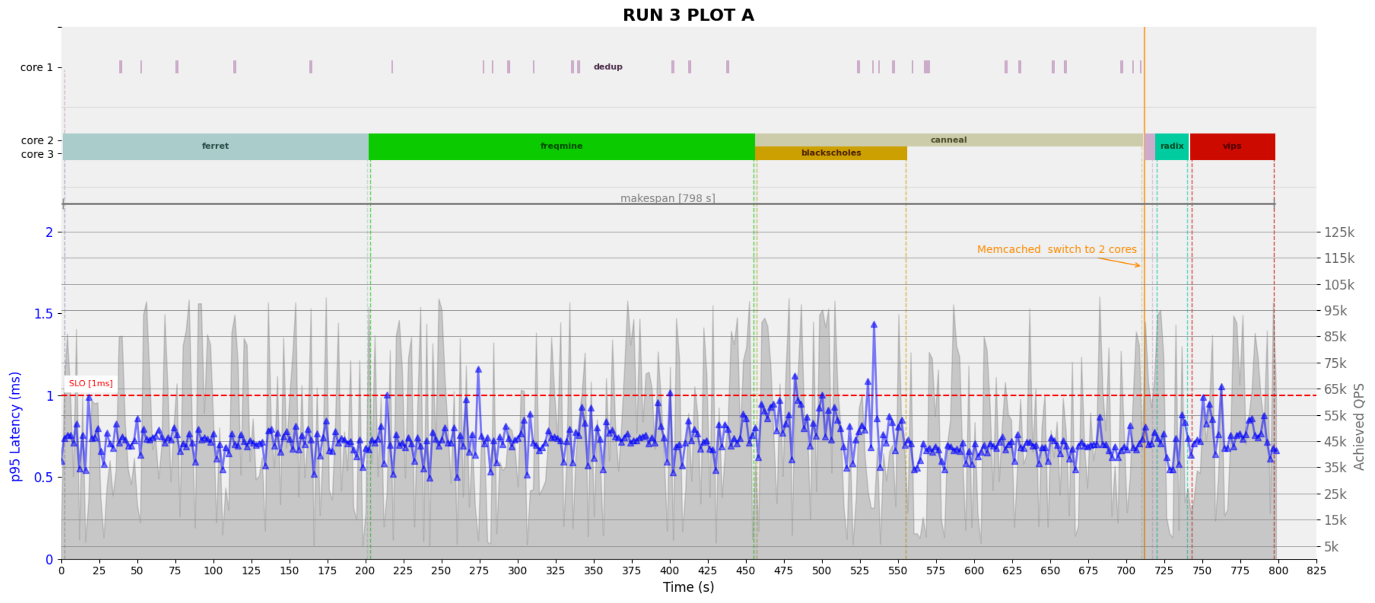

Figure 17: Plot 3A.

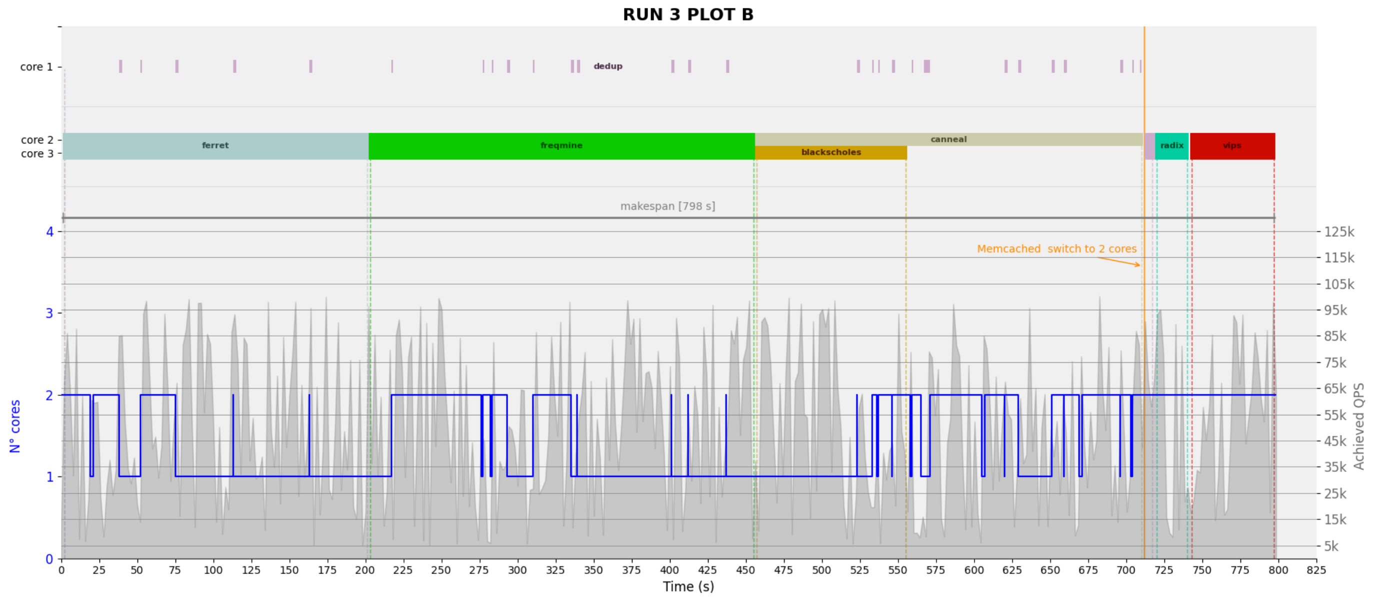

Figure 18: Plot 3B.