Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Cloud Computing Architecture

Semester project report

Group 88
Zhiyi Hu - 23-943-285
Yuan Qin - 23-955-750
Guanshujie Fu - 23-955-560

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	112.67	0.47
canneal	117.00	0.82
dedup	22.33	0.94
ferret	96.33	1.25
freqmine	154.00	0.00
radix	15.00	0.82
vips	54.33	0.47
total time	154.33	0.47

Answer: For all three runs, the SLO violation ratio for memcached is 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.

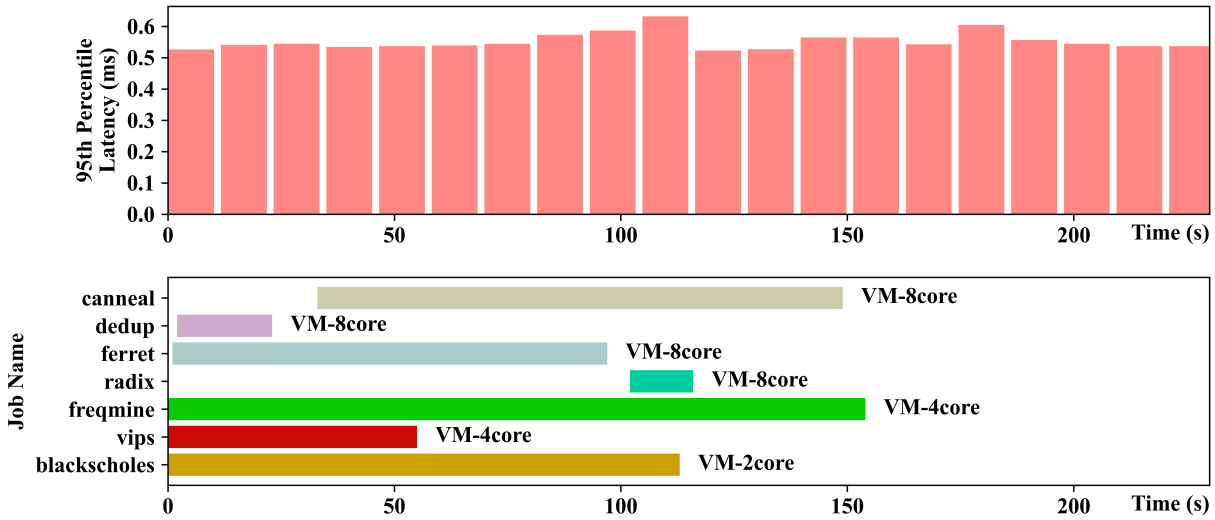
Plots: The plots are shown as figure 1.

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

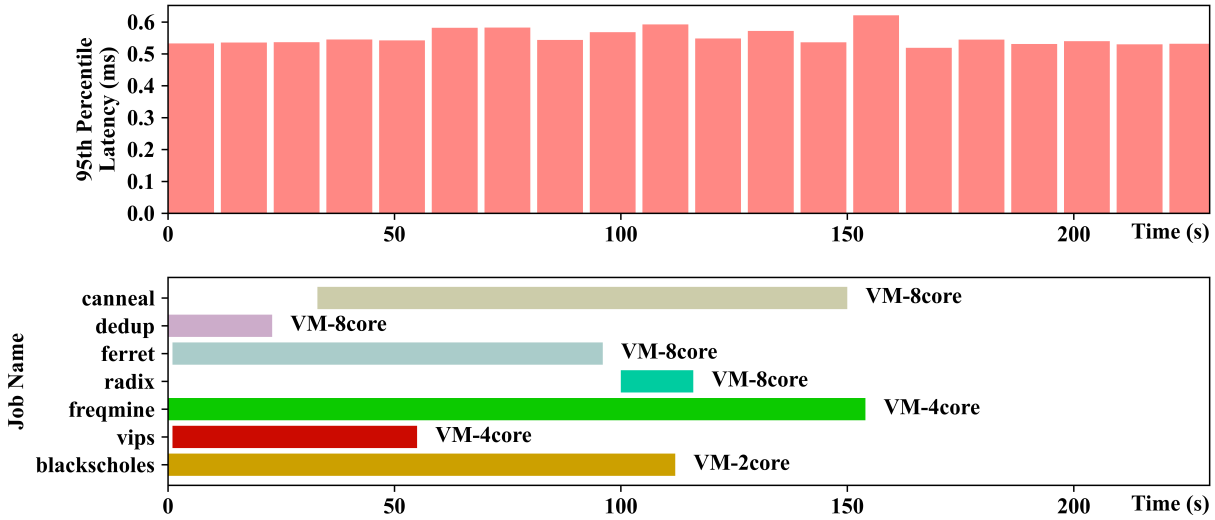
   - Which node does memcached run on? Why?

     Answer: node-a-2core. From part1 we learned that memcached is sensitive to CPU and L1i resources. By test, we found the machine with 2 cores is sufficient to guarantee the SLO and also with potential to colocate another resource-friendly job. This minimizes the potential interference of many other jobs while maximizes the resource usage.

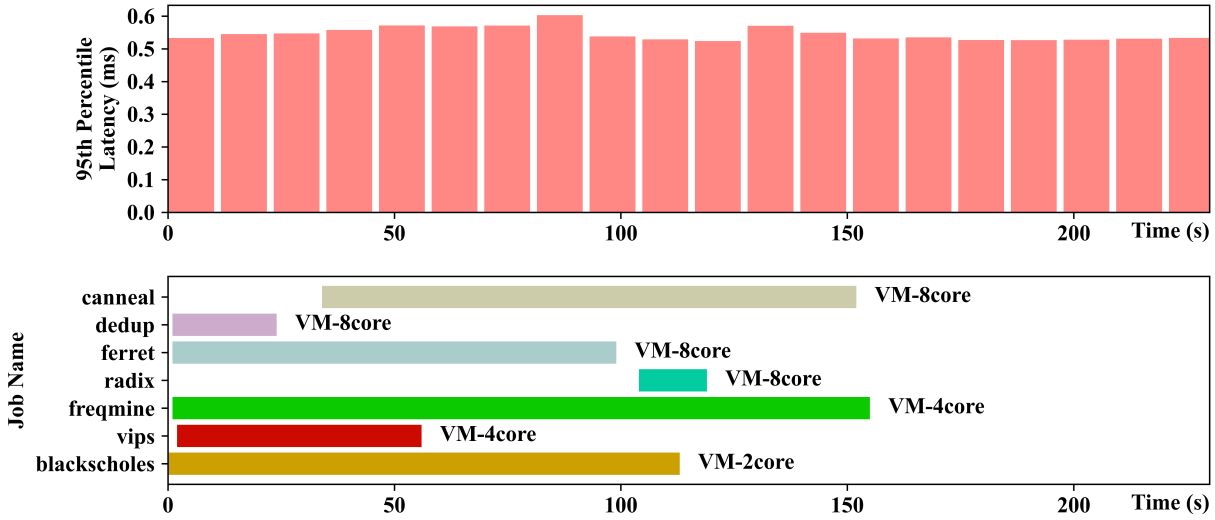
[1] Here, you should only consider the runtime, excluding time spans during which the container is paused.



95th Percentile Latency Over Time & Batch Job Execution Timeline Run 1



95th Percentile Latency Over Time & Batch Job Execution Timeline Run 2



95th Percentile Latency Over Time & Batch Job Execution Timeline Run 3

Figure 1: Benchmark Results Visualization

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

  Answer:

  - blackscholes: **node-a-2core**. Blackscholes has low CPU and LLC costs and a moderate L1i cost. This profile makes it suitable to run alongside memcached on node-a-2core, which cannot support the colocation of high-demand jobs in these resources without violating SLOs. Blackscholes is selected because, according to our experiments, other more resource-intensive jobs like ferret, freqmine, and canneal would overload node-a-2core's capabilities, while dedup, radix, and vips would underutilize this node.
  - canneal: **node-c-8core**. Canneal, which has a long runtime and high LLC demand, performs best under 8 threads. Therefore, node-c-8core, offering the most CPU and LLC resources and supporting more threads, is the ideal assignment.
  - dedup: **node-c-8core**. Dedup requires significant L1i and LLC resources, making it suitable for node-c-8core, which can provide these resources in abundance, ensuring efficient processing and minimal contention with other colocated jobs. Additionally, Dedup performs optimally with 8 threads, aligning with node-c-8core's capabilities.
  - ferret: **node-c-8core**. Ferret has the highest consumption of LLC and L1i and performs best under 8-thread configurations. Its placement on node-c-8core leverages the node's high resource availability and threading capabilities, making it a strategic fit.
  - freqmine: **node-b-4core**. Freqmine requires substantial CPU and LLC resources for its lengthy runtime, which surpasses the capacity of node-a-2core. It performs optimally with only two threads, making node-b-4core an ideal fit. Placing it on node-b-4core not only avoids conflicts with other high-demand jobs on node-c-8core but also frees up space on node-c-8core to accommodate other jobs that perform best with eight threads. This strategic distribution of resources ensures optimal job performance across nodes and maximizes utilization of node-c-8core's capabilities for jobs that require more threading.
  - radix: **node-c-8core**. With moderate resource demands and execution time, Radix is a good candidate for colocation. Its deployment on node-c-8core, which hosts multiple jobs, efficiently utilizes the available resources without causing significant contention.
  - vips: **node-b-4core**. Vips has medium resource costs across categories and a short execution time, and it performs optimally with 4 threads. Therefore, it is well-suited to node-b-4core, which offers medium resources. This strategic placement not only maximizes the node's utilization but also frees up space on node-c-8core to accommodate other jobs that require 8 threads for optimal performance. From our experiments, this strategy effectively balances the runtime between node-b-4core and node-c-8core, enhancing overall system efficiency.

- Which jobs run concurrently / are colocated? Why?

  Answer:

  - **node-a-2core**: We colocate blackscholes with memcached on this 2-core machine. As we explained above, blackscholes could maximize the utilization of resources on node-a without violating the SLO. Blackscholes runs concurrently with vips, freqmine, ferret and dedup.

- **node-b-4core**: We colocate freqmine and vips on this 4-core machine. The primary consideration is that freqmine takes the longest execution among all jobs, in which case we try to leverage this duration for concealing all other jobs' executions and also avoid deploying it with severely interfered jobs that can lead to long resource competition and severe performance degradation. We then choose vips as it has a moderate execution time with medium effects on all resources. Vips will not degrade freqmine remarkably while fully utilize the resources. Freqmine runs concurrently with all others as it lasts from the whole procedure.
- **node-c-8core**: We colocate ferret with dedup and canneal in order, canneal with radix and part of ferret in this 8-core machine. As the machine with most available resources, we found it can provide sufficient resources for concurrent jobs. We check the machine configuration and found it has 128KiB L1i cache and 55MiB L3 cache. This enables safe colocation of ferret and other 3 jobs in regardless of their conflict resource requirements and with negligible performance degradation. As we can observe, ferret runs concurrently with blackscholes, vips, frepmine, dedup and part of canneal. Canneal runs concurrently with blacksholes, vips, freqmine, ferret and radix.

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

  Order:
  Order of batch jobs in each node (local order): In node-a-2core: blackscholes. In node-b-4core: vips and freqmine start together. In node-c-8core: dedup and ferret start together, then canneal starts after dedup completes, finally, radix starts after ferret completes.
  Order of batch jobs between nodes (global order): All three nodes begin to work on batch jobs simultaneously, which means the time that blackscholes starts in node-a-2core equals the time that vips and freqmine start in node-b-4core and equals the time that dedup and ferret start in node-c-8core.

  Why: In node-a-2core, we only locate one batch job, so we start blackscholes directly at the beginning. In node-b-4core, we have allocated two batch jobs: freqmine and vips, allowing them to run concurrently instead of sequentially. This approach is taken because freqmine is a long job and during certain phases, it cannot efficiently parallelize across all four cores on that node. However, vips is relatively short. Thus, during these phases, vips can use the cores that freqmine is not utilizing, effectively reducing the total job execution time on node-b-4core. In node-c-8core, we follow the sequences that canneal starts after dedup completes and radix starts after ferret completes. Such an order can balance the total execution time of dedup and canneal with the total execution time of ferret and radix (As shown in 1, for node-c-8core, the execution of dedup and canneal overlaps with the execution of ferret and radix). As a result, there are always two batch jobs running in parallel before all batch jobs complete in that node. With each job leveraging available cores the other isn't using, the order we specify optimizes CPU cores utilization and minimizes the makespan of the batch jobs.

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

  Answer:

  - ferret: 8 threads. Ferret is a resource-intensive, long-running job that is optimized to run with 8 threads on node-c-8core. This configuration is designed to maximize parallel processing, enabling Ferret to fully utilize all available cores for the best

performance. This setup is particularly beneficial on node-c-8core because it allows Ferret to operate seamlessly during periods when all cores are available, thus maintaining high throughput and minimizing processing time. In addition, this configuration is also applied to canneal to enhance its performance, employing 8 threads for its execution.

- dedup: 4 threads. There are two key considerations: (1) Dedup is a short job that operates concurrently with ferret on node-c-8core. Ferret ideally operates with full parallelism using 8 threads to maximize the use of all 8 cores. However, due to serial components within its process, ferret cannot always maintain complete parallelism. During phases when not all cores are required for ferret, dedup can utilize these free cores. Assigning 4 threads for dedup is optimal as it allows for efficient use of available resources without compromising the core requirements for the longer-running ferret task, which generally requires the majority of the cores. (2) Based on previous findings in part2, dedup does not benefit significantly from scaling up from 4 to 8 threads, indicating that 4 threads are sufficient for optimal performance. Consequently, we use the taskset command to assign cores 0-3 to dedup. With the above configuration, we not only ensure efficient dedup operation and minimize its runtime without interfering ferret too much, but also facilitates earlier activation of canneal and frees up resources for other demanding processes.
- canneal: 8 threads. There are two key considerations: (1) In our design of the scheduler, canneal is set to run concurrently with ferret as soon as dedup completes. Given that both canneal and ferret require extended durations to complete, these processes considerably impact the total runtime. To optimize the efficiency and minimize the makespan of these workloads, we use 8 threads to enhance the parallelism of these long-running jobs on the 8-core node. (2) The execution of canneal is divided into three phases: Initially, it runs alongside ferret, competing for CPU resources. Subsequently, it operates concurrently with radix for a brief period, as radix is a pretty short task. Eventually, canneal runs alone, without any competing jobs on the node. The allocation of threads impacts the duration of each phase and we have the following trade-offs: Using 8 threads for canneal shortens the final phase. However, with the node containing only 8 cores, assigning 8 threads each to both ferret and canneal can lead to high overhead from context switching and CPU resource competition. Using fewer threads could reduce these overheads in the initial phase, allowing ferret to complete sooner, but might underutilize the CPU cores. Additionally, fewer threads would extend canneal's runtime in the final phase due to reduced parallelism. After experimenting with different configurations, we concluded that using more threads for canneal effectively reduces the overall makespan, leading us to settle on 8 threads after considering both scenarios.
- radix: 8 threads. In our design, radix runs concurrently with canneal. We also know from part 2 that radix shows little sensitivity to resource interference like CPU interference, L1i interference, and only show moderate sensitivity to LLC interference. So, canneal can cause little resource interference to radix when we run them concurrently. Radix is also a short job that consumes little resources. So, radix can cause little resource interference to canneal. Therefore, we can safely design our scheduler in order to complete radix as fast as possible and chooses the number of threads as 8 due to the good parallelism performance of radix derived in part2.
- freqmine: 8 threads. In our configuration, we have allocated two tasks (freqmine

and vips) to node-b-4core. Initially, freqmine and vips run concurrently. Vips, being a shorter task, completes quickly, allowing freqmine to then operate alone for an extended duration. The makespan, for tasks on this node is largely influenced by freqmine when it runs alone. Designed to operate with 8 threads, freqmine leverages maximum parallelism to achieve optimal performance.

- vips: 4 threads. Vips is a short task executed concurrently with freqmine on node-b-4core. As observed in interference behavior of part 2, freqmine heavily consumes CPU resources. To minimize CPU conflicts, we therefore restrict vips to fewer threads. When freqmine does not achieve full parallelism, cores remain available, making it advantageous to operate vips with multiple threads to optimize CPU usage. Based on the parallelism attributes of vips detailed in part 2, we have chosen to run it with four threads. This approach ensures efficient performance while reducing interference with other processes.
  Vips is a short job that runs in node-b-4core concurrently with freqmine, which consumes a large amount of CPU resources from the interference behavior derived in part2. So, we do not run vips with too many threads in case introducing fierce CPU interference to freqmine. In addition, the performance of freqmine is roughly the same when we run it using 4 threads or 8 threads, and there are always available cores when freqmine runs. So, we still need to run vips using multithreads so that CPUs acn be fully utilized. From parallelism property of vips derived in part2, we choose to run vips with 4 threads to ensure good performance while introducing little interference to other jobs.
- blackscholes: 2 threads. We run blackscholes in node-a-2core with one core affinity. Hence we run blackscholes using only two threads to reduce context switching overhead while also ensuring a reasonable performance.

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

  Answer: We mainly modify the YAML configuration files. There are 6 yaml files prepared. One of them configures the memcached while the rest configure the PARSEC jobs. We sort the PARSEC jobs configurations into 5 yaml files, and categorize them based on the working node and execution order. We also rename the yaml files in the format of *parsec-node-[a/b/c]-[order]* to reflect these information. In each yaml file, the main fields we modified are spec.containers.args and spec.nodeSelector. The first field is used to setup the core affinity and threads number for jobs, while the second field is used to specify working node. Moreover, we changed the spec.containers.imagePullPolicy to *Nerver* as we are trying to reduce the overhead of container cold starting period. This does not show significant effect on the results and will be changed back to *Always* in submitted version. We do not leverage any explicit Kubernetes feature. We integrate parallel job configurations into the same yaml file to guarantee the concurrent boost in the same node by Kubernetes. For instance, we put vips and freqmine in parsec-node-b-1.yaml, with ferret and dedup in parsec-node-c-1.yaml.

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

  Answer:

- **Ideas:** The primary idea of our scheduler is using the longest job to conceal other jobs, while minimizes the interference and fully utilizes the machine resources. Hence, our scheduler first starts freqmine and vips on the 4-core machine because by our test the freqmine is the bottleneck that takes the longest execution. At the same time we trigger ferret and dedup on 8-core machine with blackscholes on 2-core machine. On the 8-core machine, after dedup ends in around 20s we start canneal, and after ferret ends in 90s we start radix. As depicted in the timeline graph, freqmine spans the entire process, lasting approximately 156 seconds, while canneal ends at roughly the same time. We successfully hide the execution of other 6 jobs in the duration of freqmine.
- **Choices:** Mentioned above.
- **Trades-off:** Mentioned above.

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 2 is plotted after we averaged across three runs as recommended. The error bars in the plot represents the standard deviation of 95th percentile latency (error bars parallel to y-axis) and the standard deviation of achieved QPS (error bars parallel to x-axis).

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: First, if the number of threads used is one, the 95th percentile latency is almost the same for the configuration of one CPU core and two CPU cores: As QPS gets larger, the mean value and standard deviation of 95th percentile latency increase. But for both cases, the SLO of 1ms is violated when the target QPS is larger than 75K (corresponding to the achieved QPS of around 75K). Second, if the number of threads used is two, the 95th percentile latency varies greatly for the configuration of one CPU core and two CPU cores. For the configuration of two threads, one core, the mean value and the standard deviation of 95th percentile latency increase greatly with the increment of QPS and the SLO is violated when the target QPS is larger than 35K (corresponding to achieved QPS of around 35K). For the configuration of two threads, two cores, the mean value and standard deviation of 95th percentile latency are obviously smaller, and the 95th percentile latency keeps well below the SLO of 1ms stably when the QPS varies in our specified range. Such a difference may be due to the huge context switch overhead between two threads when we only use one CPU core and two memcached threads can achieve good parallelism and reduce 95th percentile latency of memcached when we use two CPU cores.

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 figure 2, we need two threads and two CPU cores for memcached because only this configuration of the four candidates can ensure the SLO of 1ms at the highest workload.

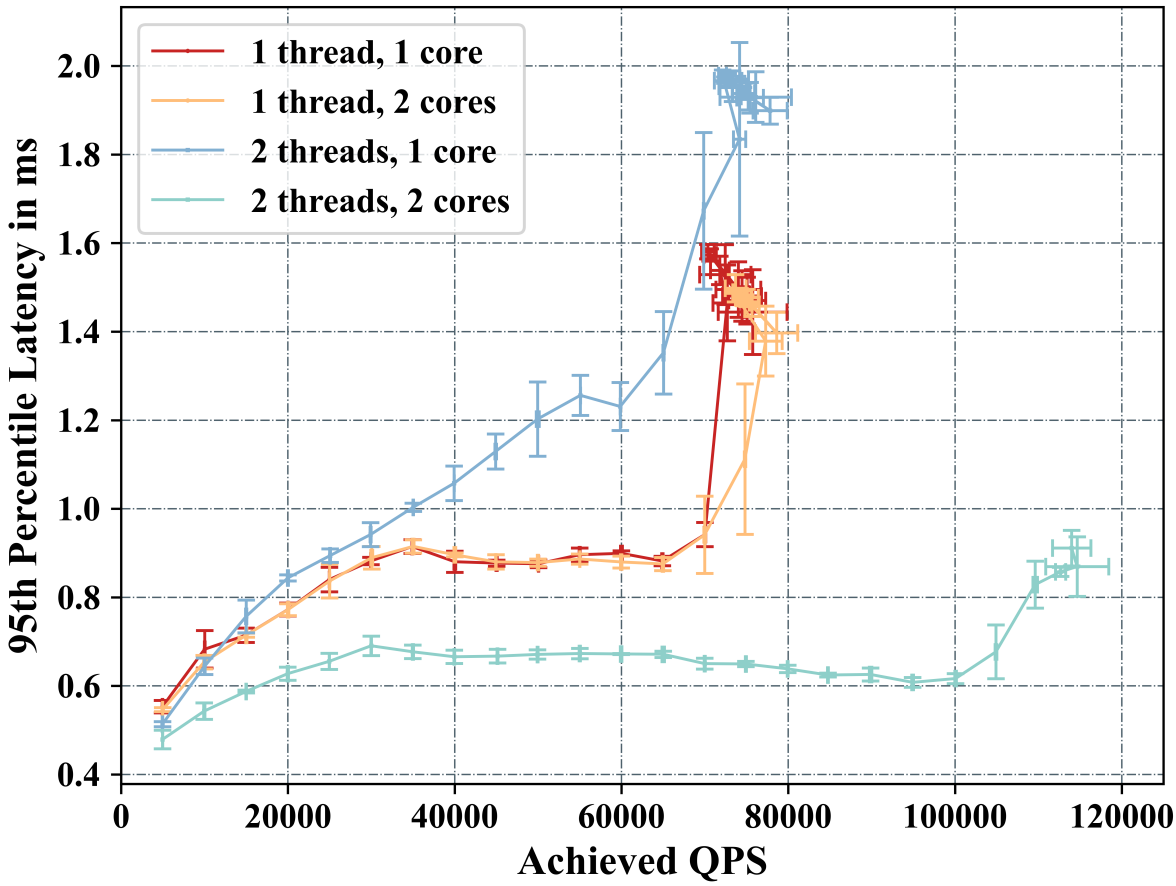


Figure 2: 95th Percentile Latency vs Achieved QPS

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

Answer: From figure 2, we need two threads and two CPU cores to ensure the SLO of 1ms at the highest workload of 125K. So, we need two threads to guarantee the 1ms 95th percentile latency SLO and change the number of CPU cores when the load varies between 5K to 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: We calculated the average results from three runs of running memcached with two threads on one core, and separately, with two threads on two cores. The corresponding plots are shown in the figure 5.

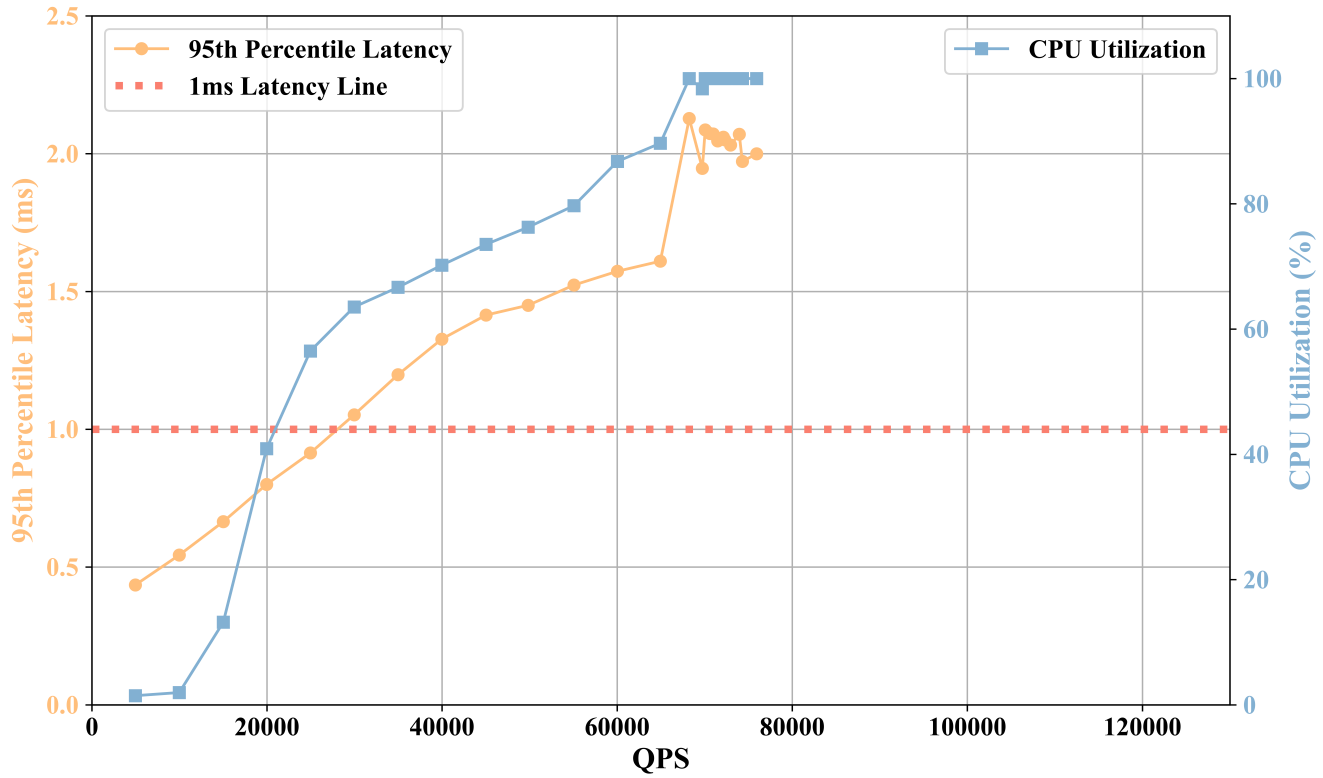
2. **[17 points]** You are now given a dynamic load trace for memcached, which varies QPS randomly between 5K and 100K in 10 second time intervals. Use the following command to run this trace:

```
$ ./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
```

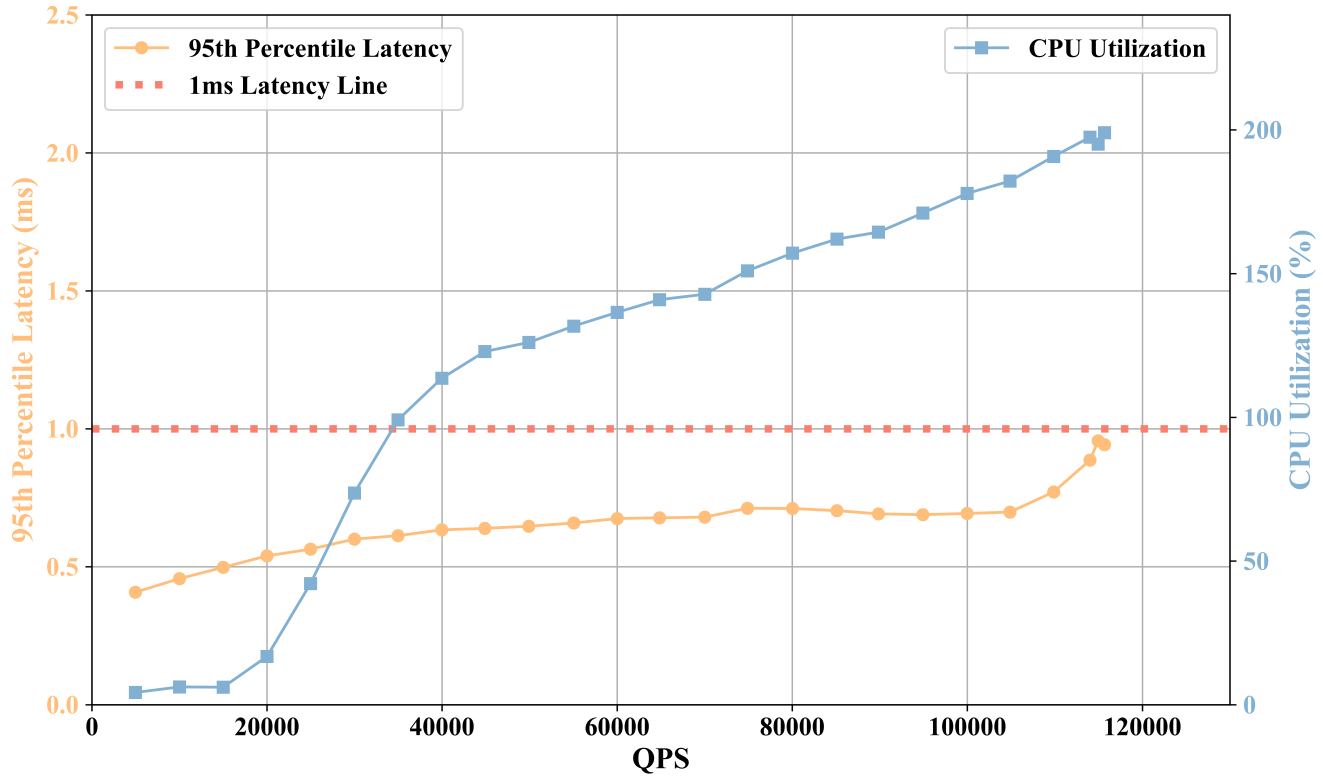
Note that you can also specify a random seed in this command using the `--qps_seed` flag.

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.



2 threads, 1 CPU core



2 threads, 2 CPU cores

Figure 3: 95th Percentile Latency vs Achieved QPS & CPU Utilization vs Achieved QPS

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 primary idea of our scheduling policy is to guarantee the priority of memcached while maximize the usage of existing resources. To guarantee the 1ms SLO, only one batch job will run concurrently with memcached, and the CPU cores will be distributed based on metrics including CPU utilization and on-time core allocation. To maximize the resources usage, we have 2 predetermined job queues, resource-intensive and resource-friendly queues, based on results from Part2. The policy will dynamically switch actively running job between these 2 queues and the switch will be triggered when some conditions are met. The scheduler will start with resource-friendly job, pop out finished jobs from corresponding queue and terminate when both queues are empty.

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

  Answer: From Part1/2, we concluded that memcached is sensitive to CPU and L1i resources. As monitoring L1i usage is difficult and meaningless, we use the information including the average CPU utilization of memcached, the instant CPU utilization of the co-running batch job, with the core number currently assigned to each of them as the metrics to dynamically determine resource allocation. Our scheduling policy starts with 4 cores for memcached, and will monitor the runtime metrics. In practice, the dynamic scheduler measures the CPU utilization of memcached every 0.1s and adjust the CPU resources for memcached based on the measurement result for the current interval ($CPU_{memcached-now}$) and previous interval ($CPU_{memcached-prev}$). With these information, we will evaluate by:

  - If $0.0\% < CPU_{memcached-now} \leq 25.0\%$ and it continues for at least 1 second, assign 1 core (0) to memcached. The low CPU utilization should imply a generally low requirements on all resources. In this case, memcached should have a low QPS that leads to a minor workload on system. Together with the results we observed from Figure 5 in Part4.1, the 1ms SLO can be guaranteed at low QPS with 1 core assigned. Hence, assigning 1 core will be safe and could provide more resources to other jobs
  - If $25.0\% < CPU_{memcached-now} \leq 50.0\%$, assign 2 cores (0-1) to memcached. Compared to previous scenario, the CPU utilization increases and the QPS should have respectively increased. From Figure 5, at least 2 cores should be required to safely guarantee 1ms SLO. Hence, the policy allocates 2 cores in this case to guarantee the SLO.
  - If $CPU_{memcached-now} > 50.0\%$ or $CPU_{memcached-now} - CPU_{memcached-prev} \geq 25\%$, assign 4 cores (0-3) to memcached. As the topmost priority is to guarantee the SLO of memcached, our scheduling policy is also tense to the increase of memcached's CPU utilization. Considering that the QPS is random and unpredictable, we will simply assign 4 cores to memcached when its CPU utilization shows significant variation.

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

Answer: As our scheduling policy does not distinguish the batch jobs to apply different adjustment, it is meaningless and a waste of time to discuss the core allocation strategy one by one. Hence, the discussion will start from the perspective of the predetermined queues mentioned above. The initial configuration of the batch jobs are stored in *jobs_init.json* including the initial core allocation, threads number and execution command. For all batch jobs, the initial CPU set is 2-3. In practice, we measure the following metrics: $CPU_{memcached-now}$ and $CPU_{memcached-prev}$ as defined above, CPU_{job} for the CPU utilization for the active job, $CoreNum_{job}$ and $CoreNum_{memcached}$ for the core number assigned to the active job and memcached process, $UtilRate_{job}$ for the active job's CPU utilization ratio calculated by $\frac{CPU_{job}}{100 \cdot CoreNum_{job}}$

- **resource-friendly** [canneal, blackscholes, radix, vips]: The scheduling policy will always start with the resource-friendly job with 2 cores (2-3).
- **resource-intensive** [dedup, ferret, freqmine]: At the beginning stage, the jobs start with 2 cores (2-3).

Based on these, we dynamically adjust the core allocation by:

- If $0.0\% < CPU_{memcached-now} \leq 25.0\%$ and $CoreNum_{memcached} < 2$, switch to resource-intensive job with 3 cores (1-3). In this case, memcached should have a minor workload on system and has been reduced to 1 core (0) affinity, it will be safe to update CPU core to 1-3 for batch job.
- If $25.0\% < CPU_{memcached-now} \leq 50.0\%$, keep executing current job and update to 3 cores (1-3). In this case, memcached still have a relatively low workload but with 2 cores (0-1) affinity.
- If $50.0\% < CPU_{memcached-now} \leq 100.0\%$, keep executing current job and increase 1 core affinity when $UtilRate_{job} \geq 0.8$ and $CPU_{memcached-now} \leq 75.0$. Otherwise the scheduler will reduce current job core affinity to 2 cores (2-3). There raises a trade-off between performance and SLO. We decide to allocate more cores for the batch job if it is starve for resources since memcached should still have a moderate QPS.
- If $100.0\% < CPU_{memcached-now} \leq 150.0\%$, switch to resource-friendly job with 1 core (3) if $UtilRate_{job} \geq 0.8$. Otherwise keep executing current job and reduce 1 core affinity if $CoreNum_{job} > 1$. There raises another trade-off between performance and SLO, and we decide to reduce the job resource as according to Figure 3 memcached should have high QPS in this scenario.
- If $150.0\% < CPU_{memcached-now} \leq 200.0\%$, keep executing current job and reduce 1 core affinity if $CoreNum_{job} > 1$.
- If $200.0\% < CPU_{memcached-now} \leq 300.0\%$, switch to resource-friendly job with 1 core (3).
- Otherwise, pause the current job. Unpause will be triggered automatically by switch/update operation in other scenario.

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

  Answer: As we can conclude from Part 2, all batch jobs demonstrate performance improvements when utilizing 4 threads. Meanwhile, given that we have a 4-core machine and our scheduling policy will allocates 2-core affinity to batch jobs for most of the time, 4 threads should be optimal for this setup. Also, using 8 threads may lead to extra context switch time while using 2 threads may not fully utilize CPU resources when it comes to 3-core affinity.

- blackscholes: 4 threads.
- canneal: 4 threads.
- dedup: 4 threads.
- ferret: 4 threads.
- freqmine: 4 threads.
- radix: 4 threads.
- vips: 4 threads.

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

  Answer: Our scheduling policy uses a sequential approach and hence no jobs will run concurrently. Because the resources on the 4-core machine is limited, to guarantee the 1ms SLO while minimizing the makespan of the batch job executions, it is unrealistic to colocate multiple jobs with memcached.

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

  Order:

  - **resource-intensive:** [dedup, ferret, freqmine]
  - **resource-friendly:** [canneal, blackscholes, radix, vips]

  Why: There does not have a specific order to execute the batch jobs in our design as our policy will dynamically switch running jobs. Meanwhile, in Part4, the runtime scenario is randomly generated and therefore no assumption will be made for the memcached QPS trace, which indicates the system workload will also randomly vary during the runtime and we cannot find a specific order for the batch jobs. Consequently, the order of execution should not theoretically affect the results, given the random variation in QPS.

  However, as we have mentioned, there are 2 predetermined queues in our scheduling policy and we order these 2 queues based on the job execution time. There is no performance consideration behind the order and any order should be feasible.

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

  Answer:

  - In Part3, the scheduling policy is static with a pre-decided settlement including the execution order, core assignment and threads allocation while the workloads can be distributed among multiple machines. However, in this policy, the strategy is dynamically adjusted based on system metrics like CPU utilization on a single 4-core machine.
  - In Part3, multiple jobs can be executed in parallel while in this part the scheduling is sequential with only one job being executed at any time point.

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

  Answer: The scheduler is implemented in Python with Docker SDK, psutil package and multi-threads.

- There are 2 threads in the scheduler. The main thread is responsible for deploying, managing and updating batch job containers while monitoring batch job CPU utilization. The secondary thread is created by main thread before batch job orchestration begins. The secondary thread is used to monitor memcached CPU utilization, adjust memcached CPU allocation and evaluate system metrics. These 2 threads will exchange information for cooperation: the main thread will deliver batch job utilization to secondary thread for evaluation and the secondary thread will fetch the evaluation back for batch job adjustment.
- For managing and monitoring batch job containers, the Python Docker SDK is leveraged. We use Docker SDK API to create/start/update/pause/unpause job containers. We also use low level API *client.api.container.stats* to calculate the CPU utilization of the active job.
- For managing and monitoring memcached process, the psutil package is used. We use *psutil.cpu_percent* for memcached CPU utilization and *psutil.cpu_affinity* to update memcached CPU core allocation.

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

  Answer:

  - **Ideas:** The high-level idea of our design has been mentioned in the overview. To maximize resource utilization, our scheduling policy incorporates a switch operation, inspired by the double buffering technique. When memcached indicates a low workload on system, resource-friendly jobs like canneal may fail to fully utilize the available CPU resources and resource-intensive jobs like ferret should come to the stage. Hence, there will be 2 container pointers *active_container* and *idle_container* that are dynamically determined during the runtime and will be exchanged when switch operation occurs. This enables fast container switch and high CPU resource utilization. Our tests reveal that the average total CPU utilization over the entire makespan is approximately 300% out of four cores.
  - **Design Choices:** In our scheduler, we utilize a multi-threading model to manage batch jobs and memcached processes in parallel. We use multi-threading instead of sequential model because of the difference in updating frequency of batch job and memcached. Our tests have shown that batch job containers should avoid high-frequency updates as these can significantly degrade performance. In contrast, memcached processes need to be monitored more frequently to instantly reflect variations due to the random QPS trace. Hence, we use 2 threads in total, with the main thread updating batch job every 0.25s and the secondary thread updating memcached every 0.1s. The update frequencies can be customized by user.
  - **Trade-offs:** Mentioned above.

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	54.80	1.92
canneal	162.82	3.17
dedup	21.00	3.89
ferret	171.14	9.55
freqmine	227.33	3.12
radix	22.30	3.14
vips	59.66	9.24
total time	721.64	14.41

Answer: The SLO violation ratios for memcached for the three runs are all 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 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: Please find the plots in Figure 4 attached in the end. We attached 6 plots that correspond to 3 runs with type A and B. For plot type A: we present QPS as lightblue bars and P95 latency with blue curve lines. The 1ms SLO is indicated by dashed gray line at the top. In the bottom part we present the execution flow for different runs. We splict the execution flow based on the predetermined queues mentioned above. The first flow reveals the execution of resource-friendly jobs including canneal, blackscholes, radix, vips, while the second flow reveals the execution of resource-intensive jobs including dedup, ferret, freqmine. The pause/unpause time points have been marked in the flows. For plot type B: the only difference is that we present the CPU core number used by memcached during the runtime with solid blue line.

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

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 did three separate runs for the load trace with a 5-second time interval, and the average total run time is 721.95s, which is slightly higher than the average total run time of 721.64s for the load trace with a 10-second time interval. The standrd deviation of total run time for these two load traces also roughly equals. So, the smaller time interval for the load to vary only insignificantly influences the execution of batch jobs, which also means the time intervals we set for the dynamic scheduler to adjust resource allocation for memcached and update the status of batch job containers are appropriate.

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: In our three separate runs for the load trace with a 5-second time interval, the SLO violation ratios are 0%, 0.56%, and 0% respectively. Compared with the results for the load trace with a 10-second time interval, the SLO violation ratio only slightly increases. As a result, our dynamic scheduler still has a good guarantee for SLO of 1ms towards high load variability.

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: We run three separate times for the load trace with a 2-second time interval, and the SLO violation ratios for each run are 1.56%, 2.00%, and 2.44% respectively. With each single run keeping the SLO violation ratio smaller than 3% and an average SLO violation ratio of about 2%, 2 seconds is the smallest `qps_interval` for our dynamic scheduler to keep the SLO violation ratio under 3%.

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

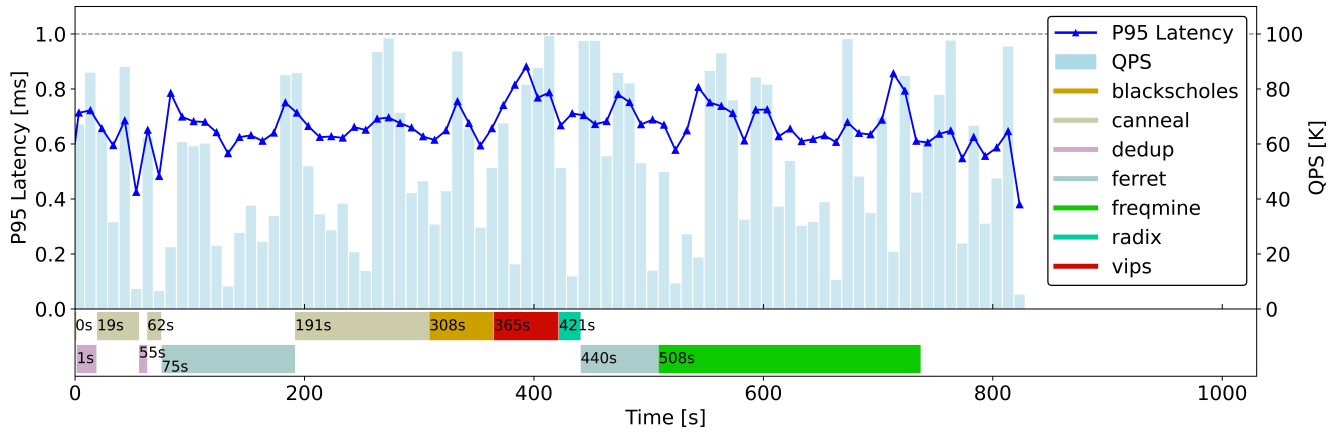
Answer: We have also tried to run for the load trace with `qps_interval` set to less than 2 seconds and find it hard to satisfy the SLO demand. For example, the SLO violation ratios for three separate runs with `qps_interval` set to 1 second are 5.33%, 5.55%, and 5.00% respectively. As a result, we think 2 seconds is the lower bound to safely keep the violation ratio under 3%. For the features that affect the minimum `qps_interval`, our suspicions are:

- As the scheduling policy in our design monitor memcached CPU usage in a static frequency, when the `qps_interval` decreases, it will be harder for the scheduler to reflect the QPS variation instantly. However, we cannot simply decrease the monitoring frequency as shorter time slice can lead to inaccurate CPU utilization measurement with glitch number like 0%.

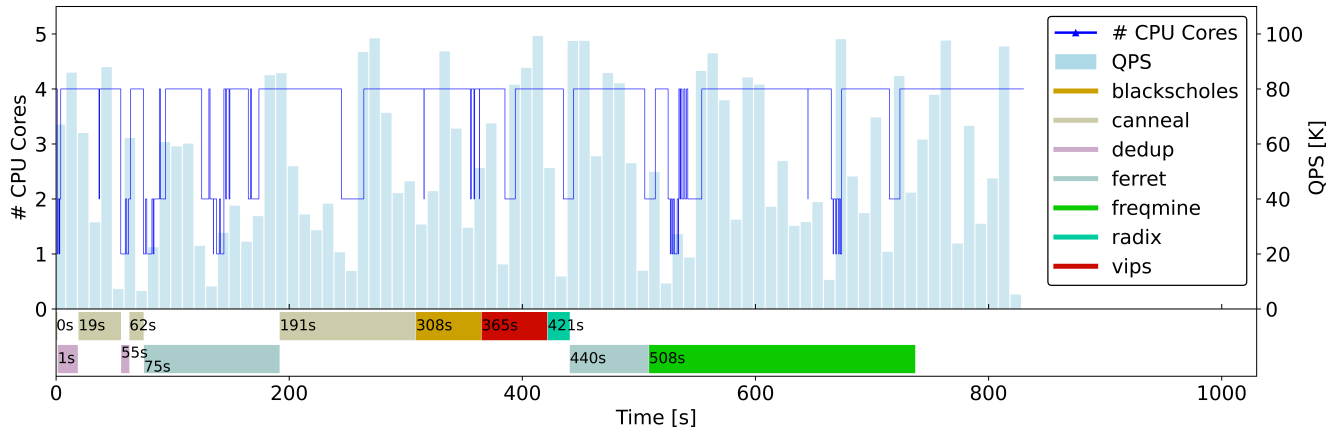
Use this `qps_interval` in the command above and collect results for three runs. Include the same types of plots (1A, 1B, 2A, 2B, 3A, 3B) and table as in Question 3.

job name	mean time [s]	std [s]
blackscholes	57.05	1.73
canneal	161.77	0.87
dedup	21.96	3.08
ferret	181.81	5.92
freqmine	221.17	1.48
radix	20.87	2.45
vips	54.84	1.19
total time	723.29	4.05

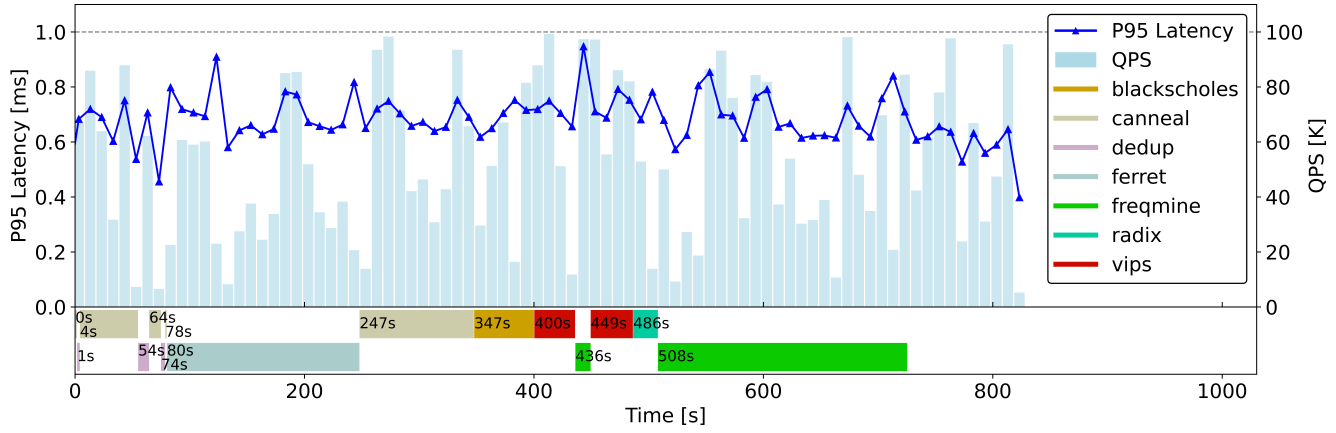
Plots: Please find the plots in Figure 5 attached in the end. Similar to Part 4.3, we attached 6 plots that correspond to 3 runs with type A and B. For this part, the plots have a much more dense QPS bars and various P95 latency, with more pause/unpause points. For the sake of clearity, we omit certain pause/unpause points that have a negligible interval.


P95 Latency [ms]
QPS [K]
Time [s]
P95 Latency
QPS
blackscholes
canneal
dedup
ferret
freqmine
radix
vips
0s 19s 62s 191s 308s 365s 421s
1s 55s 75s 440s 508s

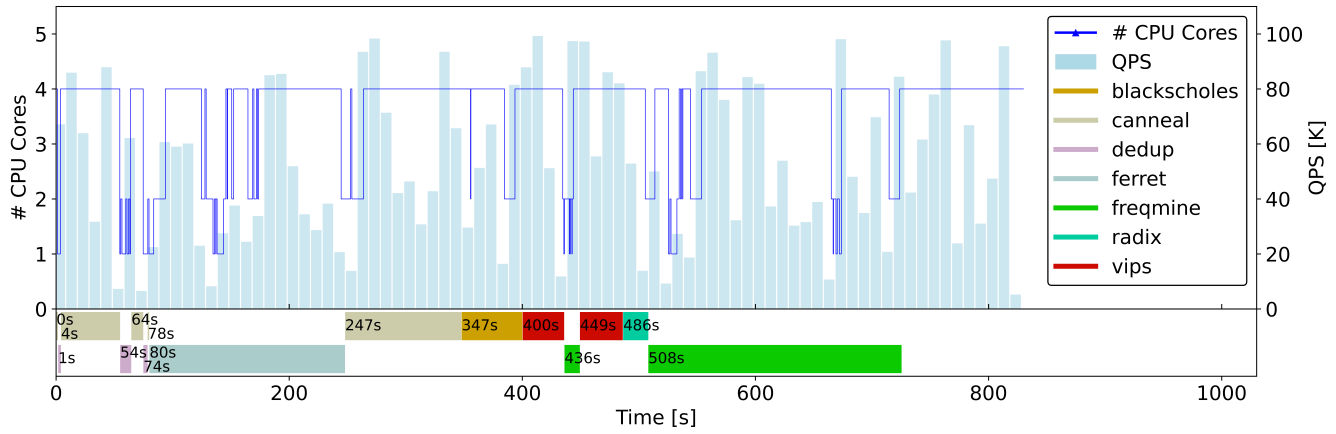
1A


CPU Cores
QPS [K]
Time [s]
CPU Cores
QPS
blackscholes
canneal
dedup
ferret
freqmine
radix
vips
0s 19s 62s 191s 308s 365s 421s
1s 55s 75s 440s 508s

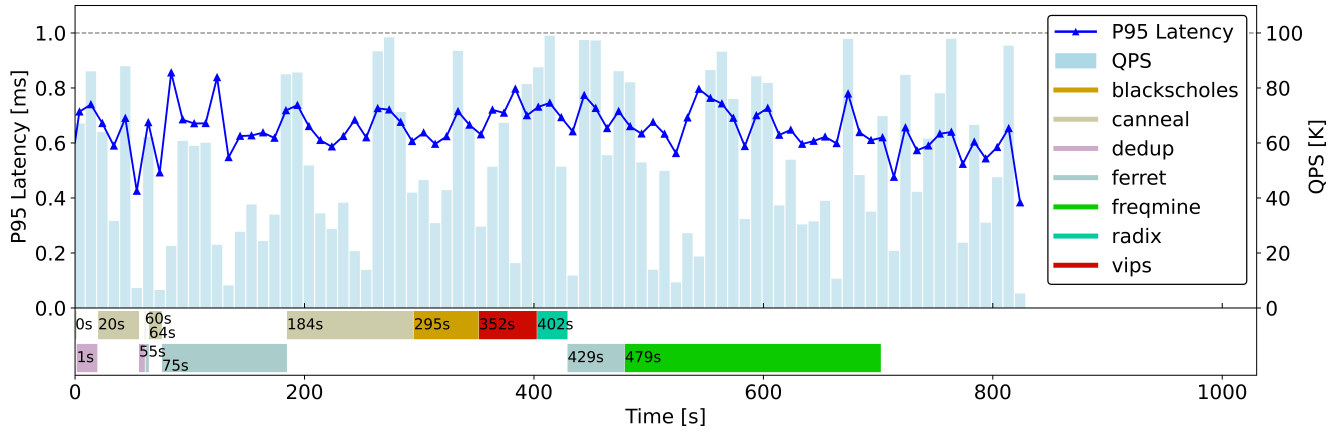
1B


P95 Latency [ms]
QPS [K]
Time [s]
P95 Latency
QPS
blackscholes
canneal
dedup
ferret
freqmine
radix
vips
0s 4s 64s 78s 247s 347s 400s 449s 486s
1s 54s 80s 74s 436s 508s

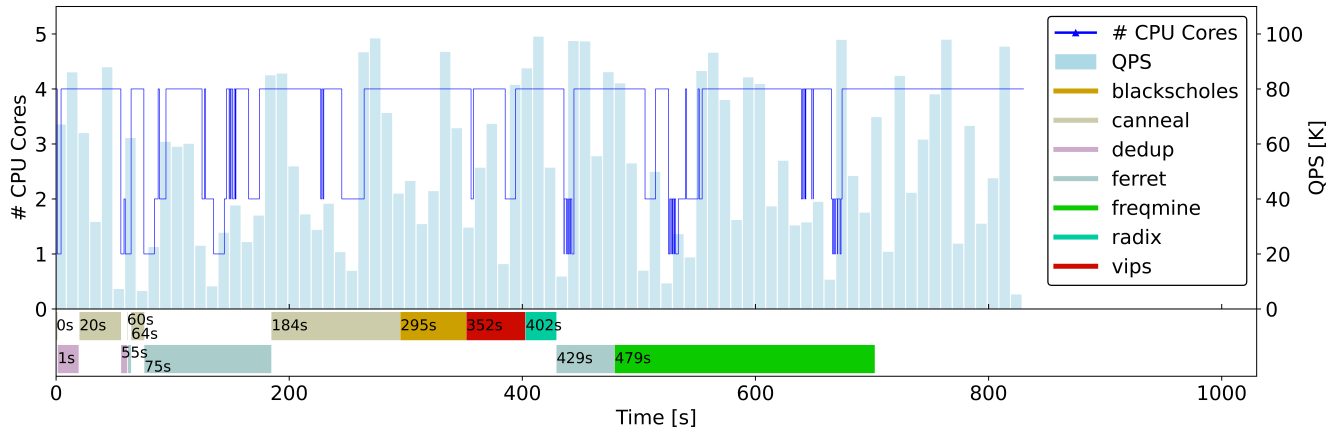
2A



2B

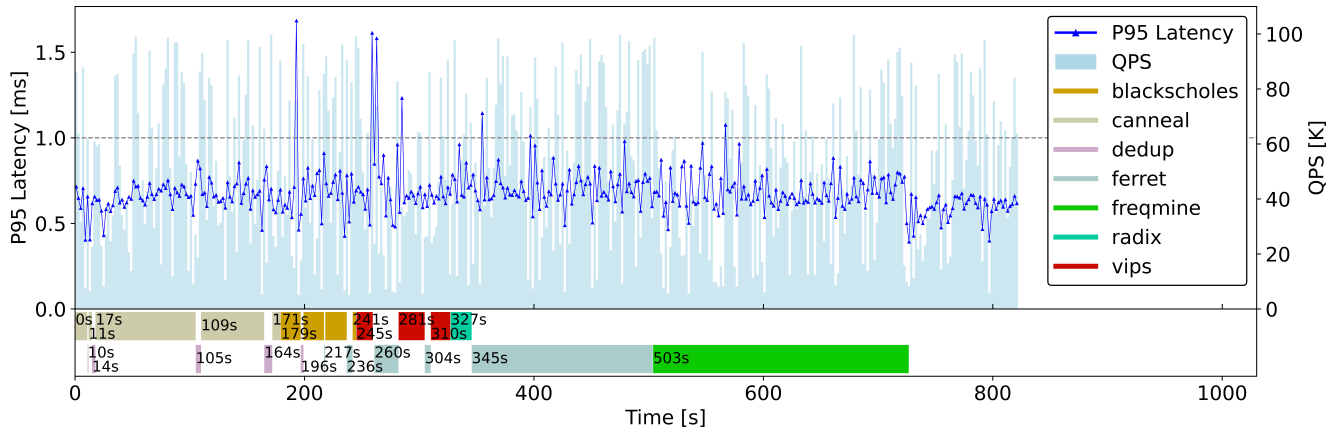


3A

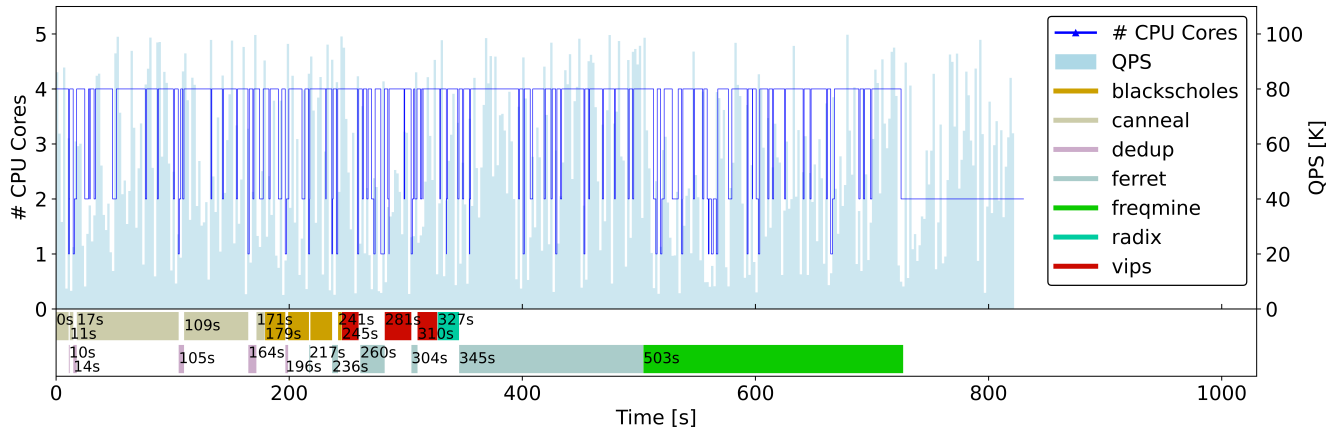


3B

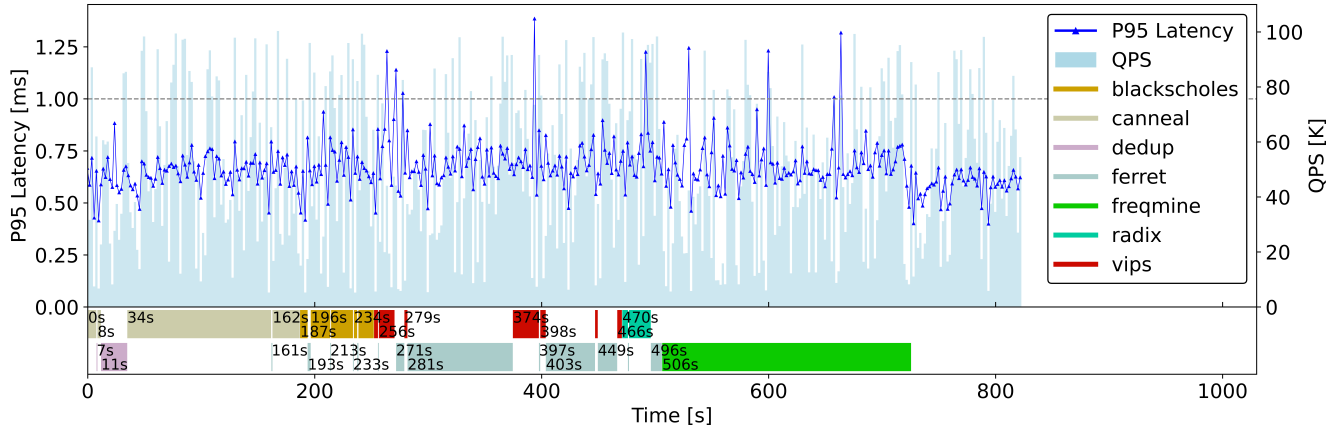
Figure 4: Plots with interval 10ms


P95 Latency [ms]
QPS [K]
Time [s]
P95 Latency
QPS
blackscholes
canneal
dedup
ferret
freqmine
radix
vips
0s 17s 11s
109s
171s 179s
241s 245s
281s
327s 310s
10s 14s
105s
164s
217s 196s 236s
260s
304s
345s
503s

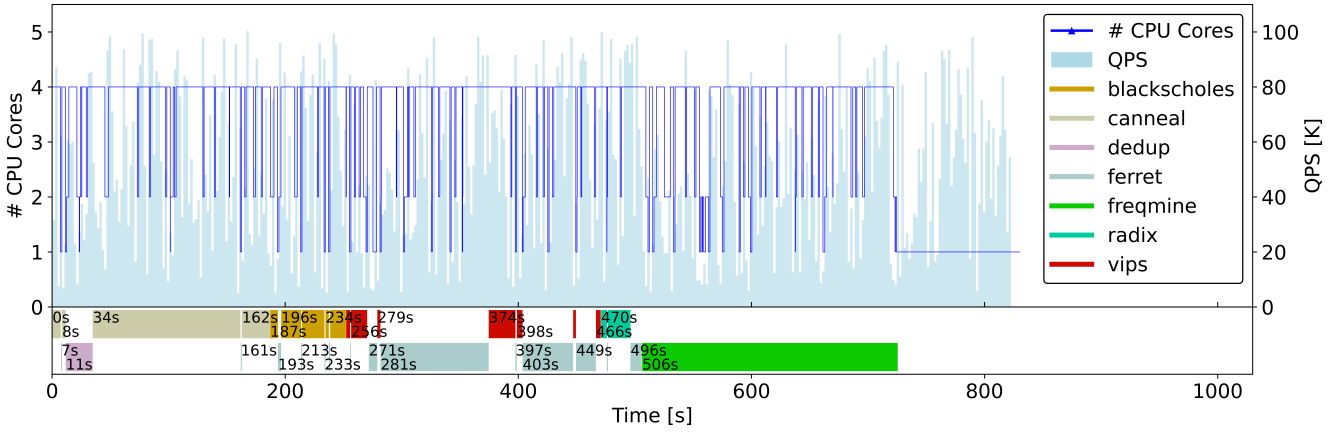
1A


CPU Cores
QPS [K]
Time [s]
CPU Cores
QPS
blackscholes
canneal
dedup
ferret
freqmine
radix
vips
0s 17s 11s
109s
171s 179s
241s 245s
281s
327s 310s
10s 14s
105s
164s
217s 196s 236s
260s
304s
345s
503s

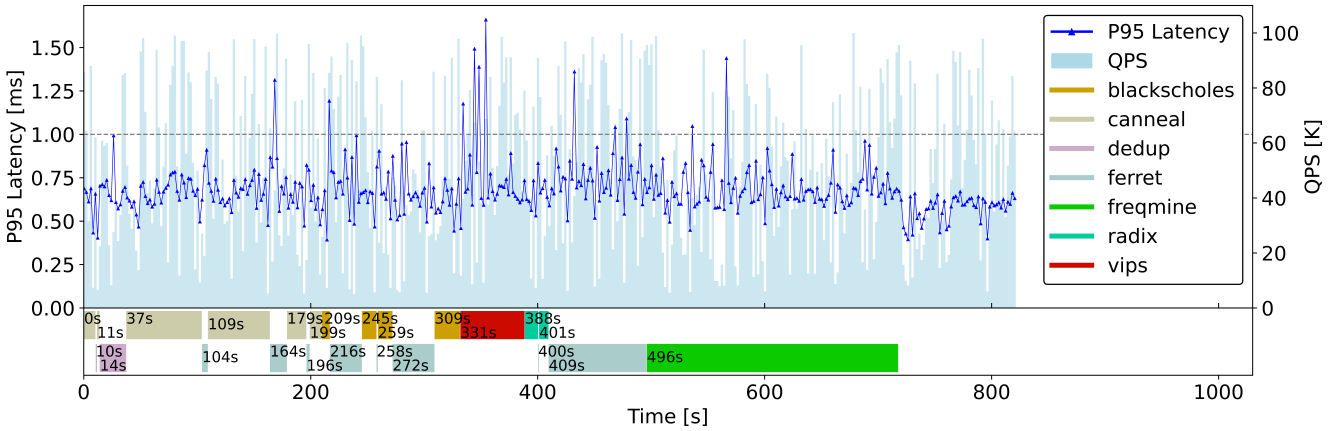
1B


P95 Latency [ms]
QPS [K]
Time [s]
P95 Latency
QPS
blackscholes
canneal
dedup
ferret
freqmine
radix
vips
0s 8s
34s
162s 187s
196s
234s
279s 256s
374s
398s
470s 466s
7s 11s
161s
213s 193s 233s
271s 281s
397s 403s
449s
496s 506s

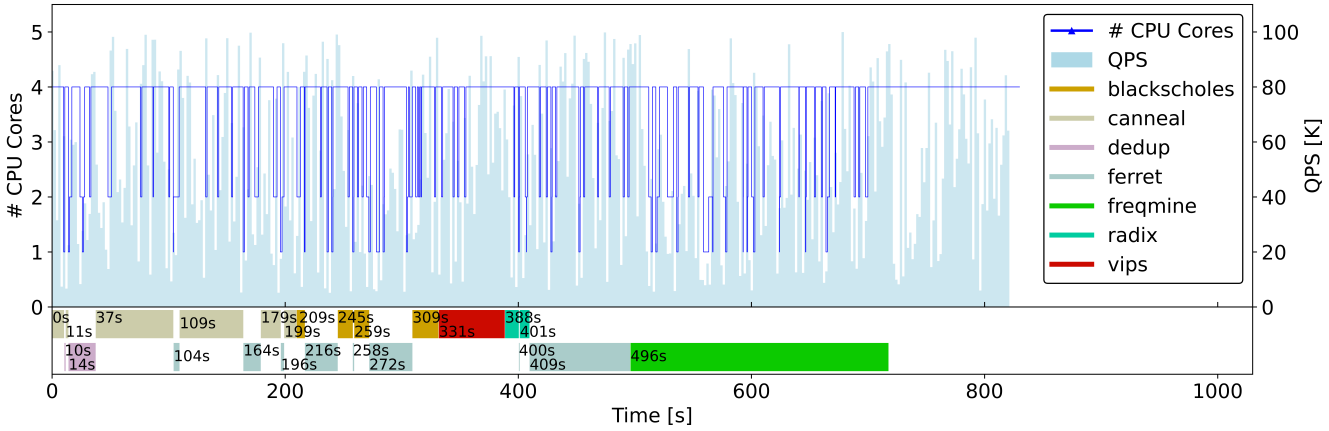
2A



2B



3A



3B

Figure 5: Plots with interval 2ms