Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Cloud Computing Architecture

Semester project report

Group 031
Matteo Pinto- Legi-NR
Soufiane Barrada - Legi-NR
Cécile Michel - Legi-NR

Systems Group
Department of Computer Science
ETH Zurich
May 11, 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.

Part 1

Using the instructions provided in the project description, run memcached alone (i.e., no interference), and with each iBench source of interference (cpu, l1d, l1i, l2, llc, membw). For Part 1, you must use the following `mcperf` command, which varies the target QPS from 5000 to 80000 in increments of 5000 (and has a warm-up time of 2 seconds with the addition of `-w 2`):

```
$ ./mcperf -s MEMCACHED_IP --loadonly
$ ./mcperf -s MEMCACHED_IP -a INTERNAL_AGENT_IP  \
           --noload -T 8 -C 8 -D 4 -Q 1000 -c 8 -w 2 -t 5 \
           --scan 5000:80000:5000
```

Repeat the run for each of the 7 configurations (without interference, and the 6 interference types) **at least 3 times** (3 should be sufficient in this case), and collect the performance measurements (i.e. the `client-measure` VM output).
Reminder: after you have collected all the measurements, make sure you **delete your cluster**. Otherwise, you will easily use up the cloud credits. See the project description for instructions how to make sure your cluster is deleted.

(a) Plot a line graph with the following stipulations:

- Queries per second (QPS) on the x-axis (the x-axis should range from 0 to 80K). (**note:** the actual achieved QPS, not the target QPS)
- 95th percentile latency on the y-axis (the y-axis should range from 0 to 6 ms).
- Label your axes.
- 7 lines, one for each configuration. Add a legend.
- State how many runs you averaged across and include error bars at each point in both dimensions.

Answer: Please refer to Figure 1.

(b) How is the tail latency and saturation point (the "knee in the curve") of memcached affected by each type of interference? What is your reasoning for these effects? Briefly describe your hypothesis.

Answer:

- `ibench-cpu`: Memcached latency is strongly affected by the cpu interference. The latency is ~5x more important with interference and explodes after the saturation point. The saturation point is around 35 000 QPS, that is half of what can be achieved without interference.

  Explanation: As we are making both the ibench-cpu and memcachd run on the same cpu, it becomes a bottleneck. As the load from the source of interference is consuming a lot of cpu resources, there is more contention and less cpu time to execute memcached instructions until the point where we can't process incoming requests anymore and drop them.

- `ibench-l1d`: Memcached latency isn't affected by the contention on the l1 data cache for the most part but has a significantly lower saturation point. The saturation point is at 55 000 QPS that is ~20% less than without interference.

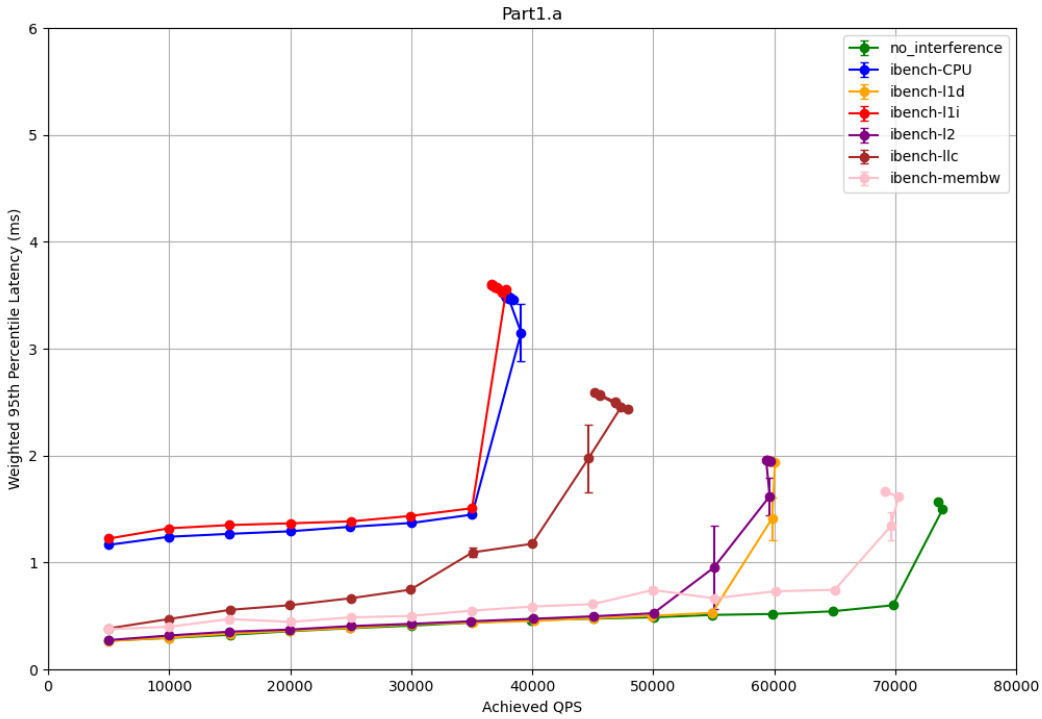

Figure 1: 95th percentile latency per achieved QPS

Explanation: For the most part the latency isn't affected by the contention, that is because memcached makes use of higher level caches which also provide the data with a low latency. However, we still get an earlier saturation point because we are still much slower than if we hit the l1d cache, which enables faster processing time per query and hence a higher throughput.

- `ibench-l1i`: Memcached latency is strongly affected by the l1i interference. The latency is ~5x more important with interference and explodes after the saturation point. The saturation point is around 35K QPS, that is half of what can be achieved without interference.

  Explanation: As we are making both the ibench-cpu and memcachd run on the same cpu, they are both contending for the same instruction cache and so it becomes a bottleneck. The instructions have to be fetched from memory, which results in a much higher latency.

- `ibench-l2`: Memcached latency isn't very much affected by the contention on the l1 data cache but has a significantly lower saturation point. The saturation point is at 50K QPS that is ~30% less than without interference.

  Explanation: For the most part the latency isn't affected by the contention, that is because memcached makes use of higher level caches which also provide the data with a low latency. However, we still get an earlier saturation point because we are still much slower than if we hit the l2 cache, which enables faster processing time per query and hence a higher throughput.

- `ibench-llc`: Memcached latency is heavily affected by the contention on the llc cache. The latency becomes gradually more important as load of client queries increases until we reach

the saturation point at 40K QPS.

Explanation: This can be explained by the fact that memcached makes use a lot of the LLC cached to serve queries rapidely, and so by contending on the llc cache, the interference causes a lot of evictions which leads to higher latency as the latency difference of fetching from the LLC cache and memory is significant.

- `ibench-membw`: The tail latency for each QPS data point is slightly higher than without interference on the memory bandwidth. The saturation point is reached somewhat sonner than without interference at around 65k QPS.

  Explanation: Because of the contention on the bandwidth, the latency for queries that require access to memory are higher, but we don't see any dramatic decrease in performance because the queries that hit in the lower level cached are not affected at all by this interference.

(c) Processor affinity:

- Explain the use of the `taskset` command in the container commands for memcached and iBench in the provided scripts.

  Answer:
  'taskset' is a command that enables choosing which CPU core to run a task on. The Linux scheduler will honor the given CPU assignment and the process will not run on any other CPUs.

- Why do we run some of the iBench benchmarks on the same core as memcached and others on a different core? Give an explanation for each iBench benchmark.

  Answer: ibench-pu because otherwise memcached and the interference will not necessarily run on the same cores and so the interference has to be much more important to have any impact. ibench-l1d, ibench-l1i, and ibench-l2 because each cpu core has it's own l1d l1i and l2 caches, so the two processes must run on the same cpu to contend for them. ibench-llc and ibench-membw run on a different core because the llc cache and the memory are shared between cores, and so to avoid any additional cpu interference, on top of the interference to the desired resource, we run them on a different cores.

(d) Assuming a service level objective (SLO) for memcached of up to 2 ms 95th percentile latency at 35K QPS, which iBench source of interference can safely be collocated with memcached without violating this SLO? Briefly explain your reasoning.

Answer: Based on our results from question a): ibench-l1d, ibench-l2, ibench-llc and ibench-membw. This is because we can clearly see for each on these intereferences that the 95th pencentile tale latency at 35K QPS is much lower than 2s. However for both ibench-cpu and ibench-l1i, the saturation point is exactly around 35K, and so it is not safe at all.

(e) In the lectures you have seen queueing theory.

- Is the project experiment above (ignoring client measure) an open system or a closed system? Explain why.

  Answer: The experiment is a closed system because we specify to the load generator (*mcperf*) the number of clients, and it's default behavior is to wait for a response before sending the next request.

- What is the number of clients in the system? How did you find this number?
  Answer: We are using the followin command to run our agent : "./mcperf -T 8 -A". This runs 8 threads, and each thread is assigned to a core. Moreover the default behavior of mcperf is to open 1 connection per thread. Hence the number of clients is 8.

- Sketch a diagram of the queueing system modeling the experiment above. Give a brief explanation of the sketch.
  Answer:

- Provide an expression for the average response time of this system. Explain each term in this expression and match it to the parameters of the project experiment.
  Answer: The average response time of the system is:

$$R = \frac{N}{X} - Z$$

  Where $N = 8$ is the number of clients. X is the throughput which is the achieved QPS. and Z is the thinking time, which in our case is 0.

(f) Below we ask you to do a theoretical cost analysis of the above experiments you ran and compare with the actual cost of your experiments, for which you will **need to track the total runtime** of the experiments, from deploying the cluster to its deletion. To find the actual cost, you can look at the **detailed billing report** after it is available on your Google Cloud project.

For the theoretical one, it would be helpful to check the official pricing calculator (link) or given individual pricing list (link). You also need to see the `machineType` and `subnets` parameters of each VM configuration by examining `part1.yaml`.

Note you can see the actual configuration details (e.g. disk size and type) of each VM configuration by looking at each individual VM instance after you deployed your cluster in `Compute Engine > VM instances > <your vm name>`.

- What is the cost per hour (in USD) for running each VM, according to the VM pricing?
  Answer: For the VM of the master node and the client-measure: 0.097118$/hour.
  For the memcachd VM: 0.071696$/hour.
  For the client-agent VM: 0.26804568$/hour.

- How much time (approximately) did your experiments take?
  Answer:

- What is the expected (theoretical) total cost and what was the actual cost of running those experiments?
  Answer: The actual cost is 1.41$

- Does the expected cost match the actual cost? (i.e. equivalent ± 0.01 if rounded up to 2 decimals). If they do not match, explain how you could achieve a more accurate estimate of the cost.
  Answer:

Part 2

1. **Interference behavior**

   (a) Fill in the following table with the normalized execution time of each batch job with each source of interference. The execution time should be normalized to the job's execution time with no interference. Round the normalized execution time to 2 decimal places. Color-code each cell in the table as follows: **green** if the normalized execution time is less than or equal to 1.3, **orange** if the normalized execution time is over 1.3 and less or equal to 2, and **red** if the normalized execution time is greater than 2. The coloring of the first three cells is given as an example of how to use the cell coloring command. **Do not change the structure of the table. Only input the values inside the cells and color the cells properly.**

Workload	none	cpu	l1d	l1i	l2	llc	memBW
blackscholes	1.00	1.46	1.49	1.79	1.45	1.72	1.55
canneal	1.00	1.90	1.95	2.38	2.09	3.41	2.37
dedup	1.00	1.88	1.69	2.31	1.52	2.46	1.86
ferret	1.00	2.63	1.71	3.08	1.67	3.63	2.74
freqmine	1.00	2.47	1.44	2.52	1.48	2.32	1.93
radix	1.00	1.12	1.17	1.19	1.18	1.75	1.21
vips	1.00	2.18	2.13	2.19	2.05	2.36	1.98

   (b) Explain what the interference profile table tells you about the resource requirements for each job. Give your reasoning behind these resource requirements based on the functionality of each job.

   Answer:

   - `blackscholes:` - Blackscholes has moderate sensitivity to all interference types, with normalized values around 1.45–1.79. This indicates it is moderately affected by CPU and cache contention. All resource types are required to some degree, but llc and l1i seems to be required the most.

     Explanation: - Blackscholes performs intensive floating-point operations over large arrays. While the operations themselves are independent (data-parallel), the data is accessed repeatedly, which stresses the instruction cache and LLC. Eviction of these cached instructions/data causes delays, and accessing DRAM (memBW) is slower.

   - `canneal:` - Canneal strongly affected by all memory-related resources and shows very high interference from cache and memory sources (L1i = 2.38, L2 = 2.09, LLC = 3.41, memBW = 2.37), with all values in the red zone.
     Explanation: - Canneal uses simulated annealing (randomly selecting elements, trying to swap them, and deciding whether the new state is "better.") with pseudorandom memory accesses, which heavily disturb cache locality. Because its data is large and unstructured, it overflows the LLC and frequently goes to main memory. Random access makes prefetching ineffective, leading to high latency under interference. LLC (Last Level Cache): Can't predict access, so cache lines are constantly evicted. memBW (Memory Bandwidth): Misses in LLC → frequent DRAM accesses. L1i (Instruction Cache): Rapid execution of many small instructions with poor locality

stresses the instruction cache too. Each thread runs independently and performs a decent amount of work.

- `dedup:` - Dedup is particularly sensitive to LLC, L1i, and CPU interference, with normalized execution times reaching up to 2.46 and 2.31. This indicates high reliance on memory and compute resources.
  Explanation: - Dedup uses a pipelined model with multiple threads working across stages that hash, compress, and check for duplicates. These stages require heavy memory operations and hashing, which increase pressure on caches (especially LLC). The hashing stages and global table access increase L1i demands and also cause contention in CPU usage.

- `ferret:` - ferret performs poorly when any resource is interfered with — especially: cpu = 2.63, l1i = 3.08, llc = 3.63 and memBW = 2.74 *Explanation:* - ferret is a pipeline-based image similarity search application. It performs a sequence of operations: segmentation, feature extraction, indexing using locality-sensitive hashing (LSH), and ranking. Each of these phases is parallelized across multiple threads, and each stage has distinct compute and memory access characteristics:
  - Segmentation and Feature Extraction: These involve complex instructions and data transformations, causing high pressure on L1 instruction cache (L1i) and CPU cycles.
  - Indexing: Uses large hash tables with LSH, resulting in poor spatial locality and frequent accesses to LLC and main memory. This increases sensitivity to LLC and memBW interference.
  - Ranking: Performs detailed similarity calculations (Earth Mover's Distance), adding further computational and memory demands.

  Because these stages run in parallel and in sequence, ferret creates constant cache evictions, instruction cache reuse failures, and memory traffic, making it one of the most interference-sensitive workloads in the suite.

- `freqmine:` - Strong impact from cpu (2.47), L1i (2.52), and memBW (1.93).
  Explanation: - Freqmine uses the FP-Growth algorithm for data mining. It constructs and recursively mines FP-trees which result in irregular data accesses and high computation. This creates a mix of CPU-heavy and memory-heavy phases, making it sensitive to both compute (cpu) and memory interference (L1i, memBW). During the main phase -data mining- the data accesses are random and different instructions are used in different phases which makes it susceptible to L1i and LCC interferences

- `radix:` - Mostly requires LLC.
  Explanation: - Radix always goes iteratively through all pages which is why LLC is important. Because the overall datasize is small enough memBW is not that important. Radix performs a radix sort which is relatively sequential and cache-friendly. The algorithm has good locality, so even under interference it handles well unless LLC is under extreme pressure. Small enough data size also helps avoid heavy memory usage.

- `vips:` - High sensitivity: LLC (2.36), L1i (2.19), memBW (1.98).
  Explanation: - VIPS is an image processing pipeline with multiple stages. Each stage involves reading/writing image segments, performing transformations, and applying filters. Large image sizes stress the cache and memory system. Although some stages are parallelized, performance depends on maintaining pipeline flow — which stalls under cache and bandwidth interference.

(c) Which jobs (if any) seem like good candidates to collocate with memcached from Part 1, without violating the SLO of 2 ms P95 latency at 40K QPS? Explain why.

Answer: Radix is a good candidate to collocate with memcached without violating the 2 ms P95 latency SLO at 40K QPS. This is because it mainly uses the LLC and shows low interference in CPU, L1i, and memBW—the components that have the most impact on latency. As Figure 1 shows, memcached tolerates LLC interference at 40K QPS, keeping P95 latency under the SLO.

2. **Parallel behavior**

(a) Plot a line graph with speedup as the y-axis (normalized time to the single thread config, Time_1 / Time_n) vs. number of threads on the x-axis (1, 2, 4 and 8 threads - see the project description for more details). Pay attention to the readability of your graph.

Answer: Please refer to Figure 2.

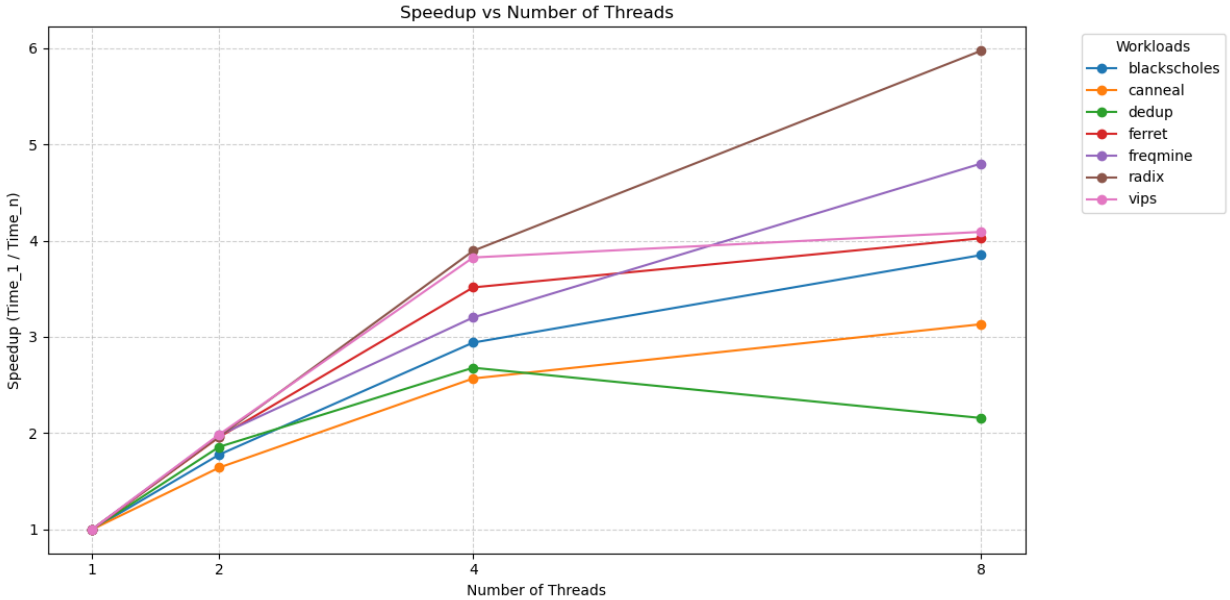

Figure 2: Speedup vs Number of threads

(b) Briefly discuss the scalability of each job: e.g., linear/sub-linear/super-linear. Do the applications gain significant speedup with the increased number of threads? Explain what you consider to be "significant".

Answer:

- blackscholes:
- canneal:
- dedup:
- ferret:
- freqmine:
- radix:
- vips:

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	67	1
canneal	189	4.36
dedup	28.33	0.58
ferret	179.33	0.58
freqmine	174	1
radix	23	0
vips	104	0
total time	189	4.36

Answer: For each run, we define $t_{i,j}$ where $i \in \{\text{blackscholes, canneal, dedup, ferret, freqmine, radix, vips}\}$ is the job name and $j \in \{1, 2, 3\}$ the run index. Here are the formulas for the mean time and standard deviation:

$$\text{mean}_i = \frac{t_{i,1} + t_{i,2} + t_{i,3}}{3}$$

$$\text{std}_i = \sqrt{\frac{(t_{i,1} - \text{mean}_i)^2 + (t_{i,2} - \text{mean}_i)^2 + (t_{i,3} - \text{mean}_i)^2}{2}}$$

To evaluate Memcached latency performance, we define:

- **Start time (T_{start}):** The earliest start time among all batch jobs (excluding memcached).
- **End time (T_{end}):** The latest end time among all batch jobs (excluding memcached).
- N_{total}: total number of latency data points recorded during $[T_{start}, T_{end}]$
- $N_{violations}$: number of data points where p95$^{\text{th}}$ latency > 1ms.

The **SLO Violation Ratio** is computed as:

$$\text{SLO Violation Ratio}_{memcached} = \frac{N_{\text{violations}}}{N_{\text{total}}} = 0 \quad (1)$$

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

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:

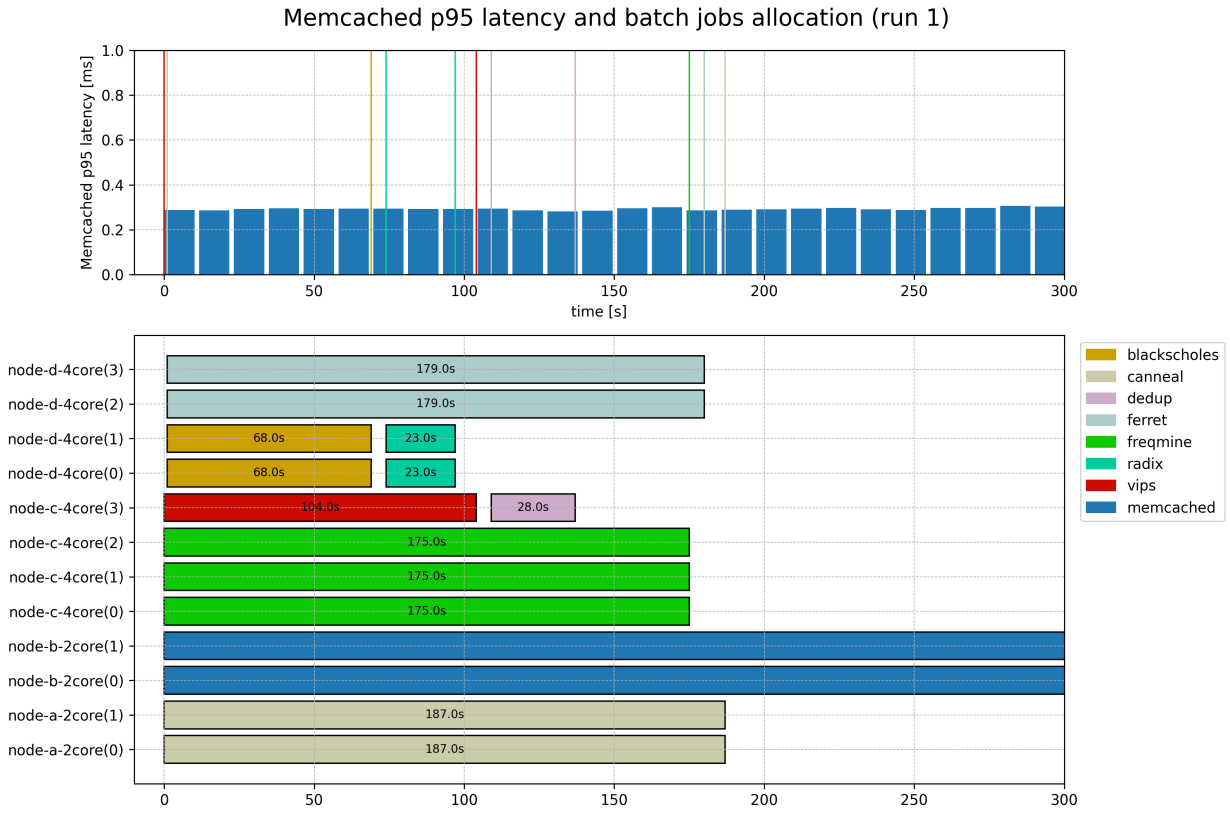

Figure 3: This figure shows the measurements of the first run. **The upper diagram** displays the p95 latency of `memcached` in each measurement interval using the augmented version of `mcperf`. Each bar represents the p95 latency at a specific time slice. The vertical colored lines indicate the exact start and end times of each batch job. **The lower diagram** visualizes the mapping of batch jobs to physical cores across nodes during execution. Each horizontal bar shows where and for how long a specific batch job ran, with the job name represented by both color and bar label. The y-axis labels show the specific cores across nodes (e.g., `node-d-4core(1)`). For example, `freqmine` was scheduled on three cores of `node-c`, running concurrently, while `ferret` occupied two cores on `node-d`. Jobs such as `radix` and `blackscholes` were scheduled sequentially on the same core, one after the other. **Note** that some jobs, such as `freqmine` and `vips`, were executed simultaneously, making their corresponding vertical lines in the upper plot difficult to distinguish due to overlapping.

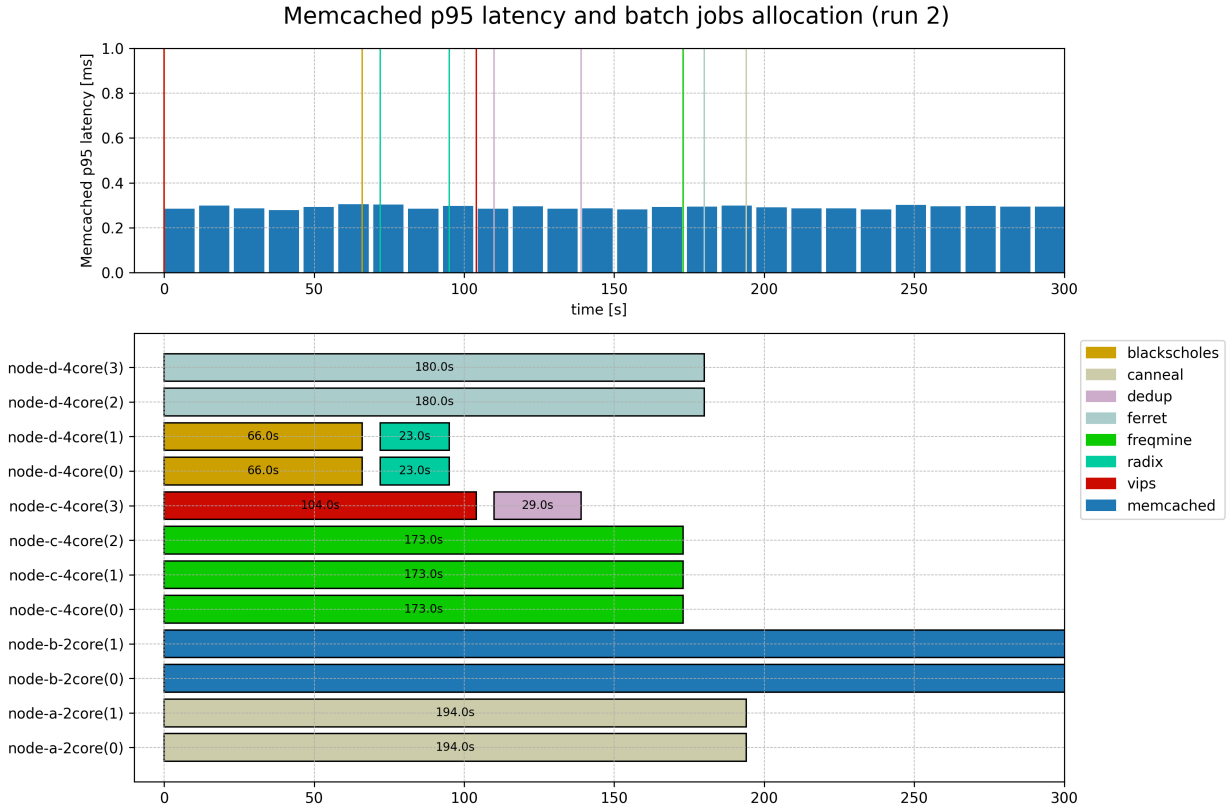

Figure 4: This figure shows the measurements of the second run. **The upper diagram** displays the p95 latency of `memcached` in each measurement interval using the augmented version of `mcperf`. Each bar represents the p95 latency at a specific time slice. The vertical colored lines indicate the exact start and end times of each batch job. **The lower diagram** visualizes the mapping of batch jobs to physical cores across nodes during execution. Each horizontal bar shows where and for how long a specific batch job ran, with the job name represented by both color and bar label. The y-axis labels show the specific cores across nodes (e.g., `node-d-4core(1)`). For example, `freqmine` was scheduled on three cores of `node-c`, running concurrently, while `ferret` occupied two cores on `node-d`. Jobs such as `radix` and `blackscholes` were scheduled sequentially on the same core, one after the other. **Note** that some jobs, such as `freqmine` and `vips`, were executed simultaneously, making their corresponding vertical lines in the upper plot difficult to distinguish due to overlapping.

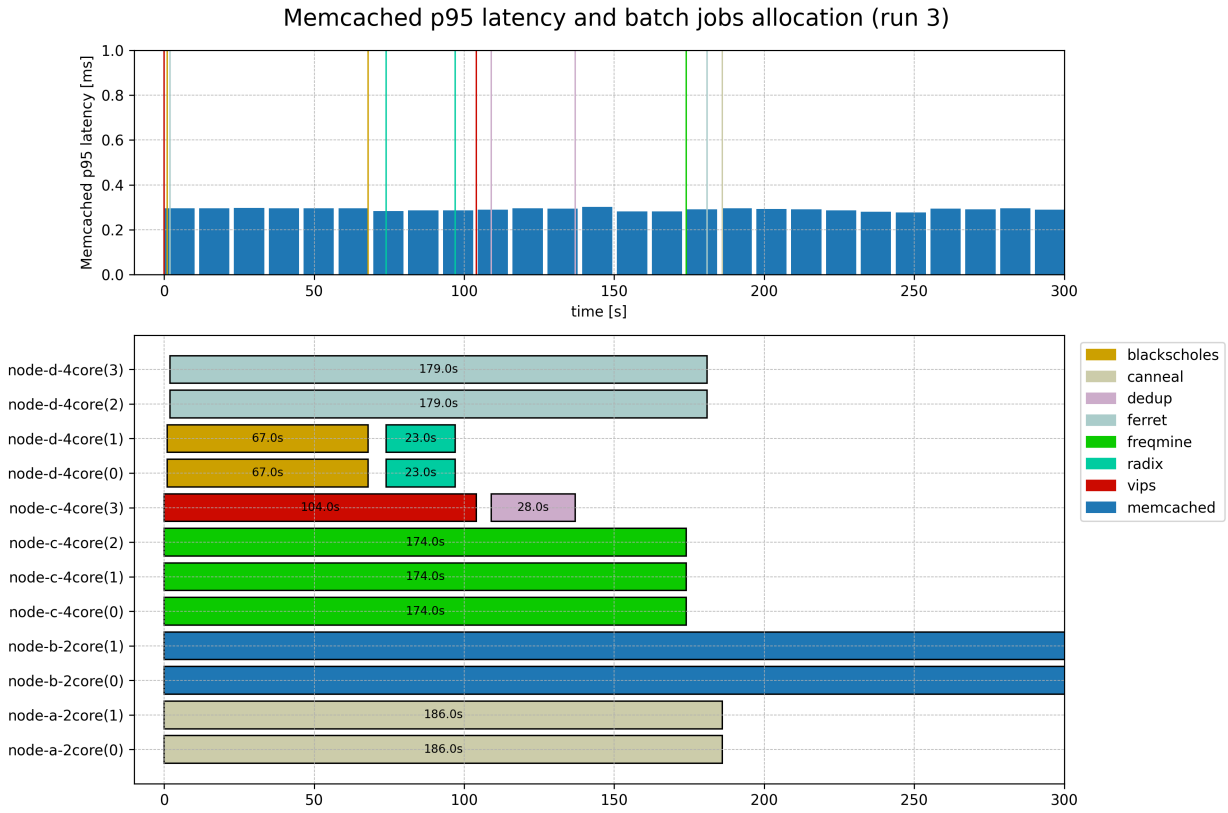

Figure 5: This figure shows the measurements of the third run. **The upper diagram** displays the p95 latency of `memcached` in each measurement interval using the augmented version of `mcperf`. Each bar represents the p95 latency at a specific time slice. The vertical colored lines indicate the exact start and end times of each batch job. **The lower diagram** visualizes the mapping of batch jobs to physical cores across nodes during execution. Each horizontal bar shows where and for how long a specific batch job ran, with the job name represented by both color and bar label. The y-axis labels show the specific cores across nodes (e.g., `node-d-4core(1)`). For example, `freqmine` was scheduled on three cores of `node-c`, running concurrently, while `ferret` occupied two cores on `node-d`. Jobs such as `radix` and `blackscholes` were scheduled sequentially on the same core, one after the other. **Note** that some jobs, such as `freqmine` and `vips`, were executed simultaneously, making their corresponding vertical lines in the upper plot difficult to distinguish due to overlapping.

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

   - Which node does memcached run on? Why?

     Answer: memcached runs on the node-b. The reason is that even though it doesn't provide the best cpu resources nor a lot of memory, we concluded from Task 1 that it is largely capable of meeting the SLO requirements unless there is a high CPU or l1i interference, which we avoid. This enables us to keep the other better performing cores and higher memory nodes for other more resource hungry tasks.

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

Answer:

- blackscholes: Blacksholes runs on node-d. Since it neither suffers from nor causes significant interference, it is ideal for collocating with other jobs. We considered both node-c and node-d for its deployment, and chose node-d because Blacksholes isn't CPU-intensive and wouldn't benefit from the more powerful CPU available in node-c.
- canneal: canneal runs on node-a. canneal is both memory and cache dependent, but also sensitive to l1i interference. So, we decided to run it on it's own on a high-memory node, using the fact that it scales decently with parallelism. It doesn't need the best performing CPU, so node-a was the right fit.
- dedup: dedup runs on node-c. We mainly put it here because it's a short running job that only suffers from sharing a core. So we could just run it after vips finishes.
- ferret: ferret runs on node-d. ferret is a long running job that benifits a lot from parallelism (until 4 threads), and also is at the same time CPU and memory hungry, so node-d presented the perfect balance for all of this.
- freqmine: freqmine runs on node-c. freqmine is a long running job that benifits from parallelism, and so we put it on 3 cores. Also it is CPU hungry, and node-c provides the best performing CPUs.
- radix: radix runs on node-d. radix is a small job, that doesn't require much resources and doesn't interfere with other jobs, so we could just run it after blackscholes finishes running.
- vips: vips runs on node-c. vips needs a better CPU, and it experimentally did fine when running on the same node as freqmine, in the sense that the slowdown wasn't that important.

- Which jobs run concurrently / are colocated? Why?

  Answer: ferret runs colocated with blackscholes and then afterwards with radix. ferret is the most sensitive job from all the proposed ones, and so we made sure to run it with the 2 least invasive jobs that are blacksholes and radix. On the other hand, freqmine runs colocated with vips and then dedup, the reason is that even though they might contend for llc cache, vips and dedup are relatively small jobs, and they are the most cpu hungry jobs so we wanted them to run on the best CPU cores of node-c. In other configurations we tried, we saw that canneal, which currently constitutes our bottlneck, could be run much faster (2x) by using the 4 cores on node-c, but we get a comparable makespan at the end because of a new bottleneck.

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

  Order (to be sorted): [blackscholes, ferret, freqmine, vips, canneal] ; [dedup, radix]

  Why: We run all the longer jobs from the beginning, and as dedup and radix are comparatively very short, we run them just after the medium running jobs blacksholes and vips. We also explored running radix and dedup from the beginning in colocation with memcache, but it failed due to OOM, and also the node-d wasn't the bottleneck anymore after some changes so radix run just fine in colocation.

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

Answer: Experiments showed us that using more threads than the number of allocated cores for a job doesn't result in any performance gain, perhaps because of the overhead of context switching. Hence, we allocate each time the same number of threads as the number of allocated cores.

- blackscholes: 2
- canneal: 2
- dedup: 1
- ferret: 2
- freqmine: 3
- radix: 2
- vips: 1

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

  Answer: We make use of a script that adapts each time it is run all the YAML files according to the new allocation we specified. More specifically, we populate in our script, for each job, the node we want it to be run on, the cores it should run on using taskset, and the number of threads. We didn't make use of any additional Kubernetes features.

- 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 allocated the jobs on nodes based on the insights gained from task 1 and task 2, and then we experimented with and without concurrency for some setups. More Specifically we saw in the beginning that Radix was a rather low-weight job and decided to run it in colocation with memcached but this has generated an OOM error so we transfered it to another node that we saw had some idle period. Then seeing that canneal is very hard to colocate with any other job because of how sensitive it is to most types of interferences, we decided to run it on it's own. Canneal constitutes the botlleneck of our scheduling, so in one of our schedulings we experimented running it on node-c on the 4 cores, which led to a huge speed up, but due to the other changes to the scheduling we had to make, we ended up at roughly the same makespan". We previously already explained how we ended up choosing the allocation for ferret along with blacksholes. As for freqmine, colocating it with vips and dedup led to a slowdown of approximately 25 seconds compared to when we run it alone on 4 cores in our other scheduling policy, and we thought that this is ok, on top of the fact that it's not the botlneck. Compared to our Second best run where we run everything concurrently, there isn't much difference in the makespan. However, what was interesting is seeing that our hypothesis of canneal and ferret running well together concurrently on node-c did actually yield to good results, as they were far from being the bottleneck. Our rationale was that since both jobs are frequently accessing memory, they often pause processing while waiting for data. Concurrency could potentially allow one job to utilize the CPU while the other waits for memory, improving the overall throughput. In a slightly different version of our second best run where we tried to improve it, we tried exploiting 1 of the cores in node-b (as we know from task1 that we only need 1 core to satisfy the SLO for

memcached), and used it to ran vips, but it ended up being the botteleneck and had approximately the same makespan.

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

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

Part 4 [74 points]

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

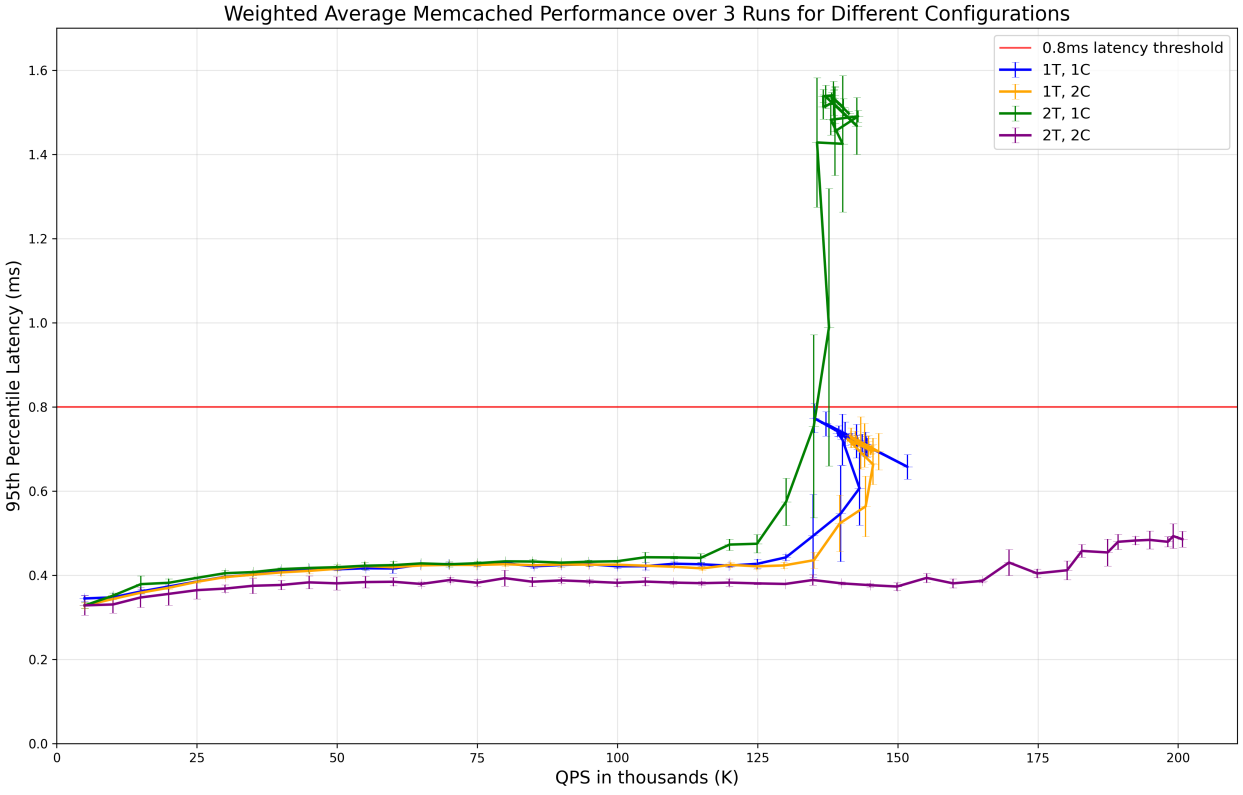

Figure 6: Weighted average 95th percentile latency vs. achieved QPS for each target QPS across three runs. Latency values are weighted by achieved QPS, and standard deviation is shown with error bars. Each line represents one of the tested Memcached configurations.

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: Memcached performs best with 2 threads and 2 cores, sustaining high throughput (up to 200K QPS) with low and stable latency, clearly outperforming other configurations. The 1-thread setups (1T, 1C and 1T, 2C) show similar behavior, meeting the 1ms latency SLO up to around 140K QPS. In contrast, the 2-thread, 1-core configuration quickly exceeds the SLO beyond 125K QPS, highlighting the performance penalty of thread contention on a single core.

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: To support the highest load in the trace (approximately 220K QPS) without violating the 0.8ms latency SLO, the only configuration that satisfies both requirements is T = 2 threads and C = 2 cores. While the 1T configurations (1T, 1C and 1T, 2C) remain under the 0.8ms latency threshold, they plateau at around 140K QPS and cannot sustain higher throughput. The 2T, 1C violates the SLO at around 140000 due to contention. Therefore, 2 threads and 2 cores is the only configuration that achieves both sufficient throughput and acceptable latency.

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: To guarantee the 0.8ms 95th percentile latency SLO while the load varies between 5K and 220K QPS, we have to use $T = 2$. The 1-thread configurations either plateau or, in the case of 1T, 2C, eventually violate the SLO. With dynamic core allocation, starting from $C = 1$ and scaling up to $C = 2$ as load increases (around 140K) is sufficient to guarantee the 0.8ms 95th percentile latency 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 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:

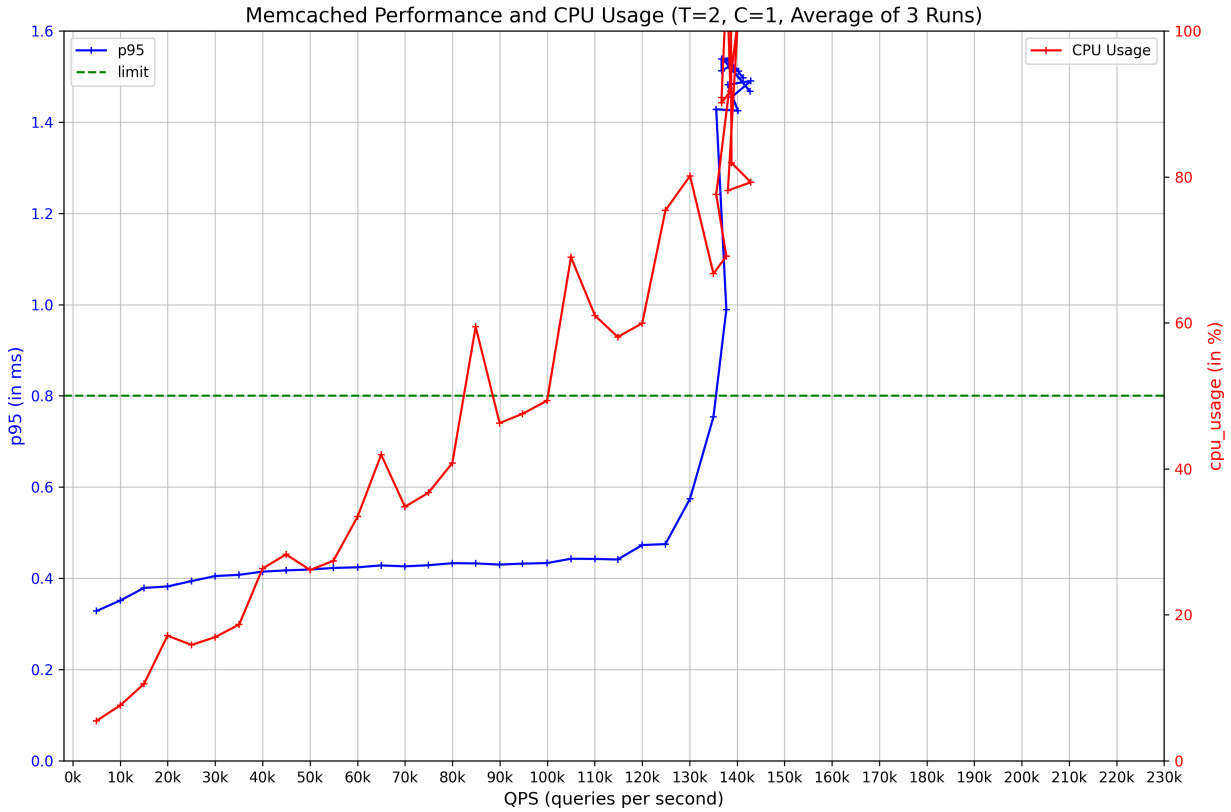

Figure 7: Weighted Average p95 Latency and CPU Utilization vs. Achieved QPS in Memcached (T=2, C=1, Averaged Over 3 Runs)

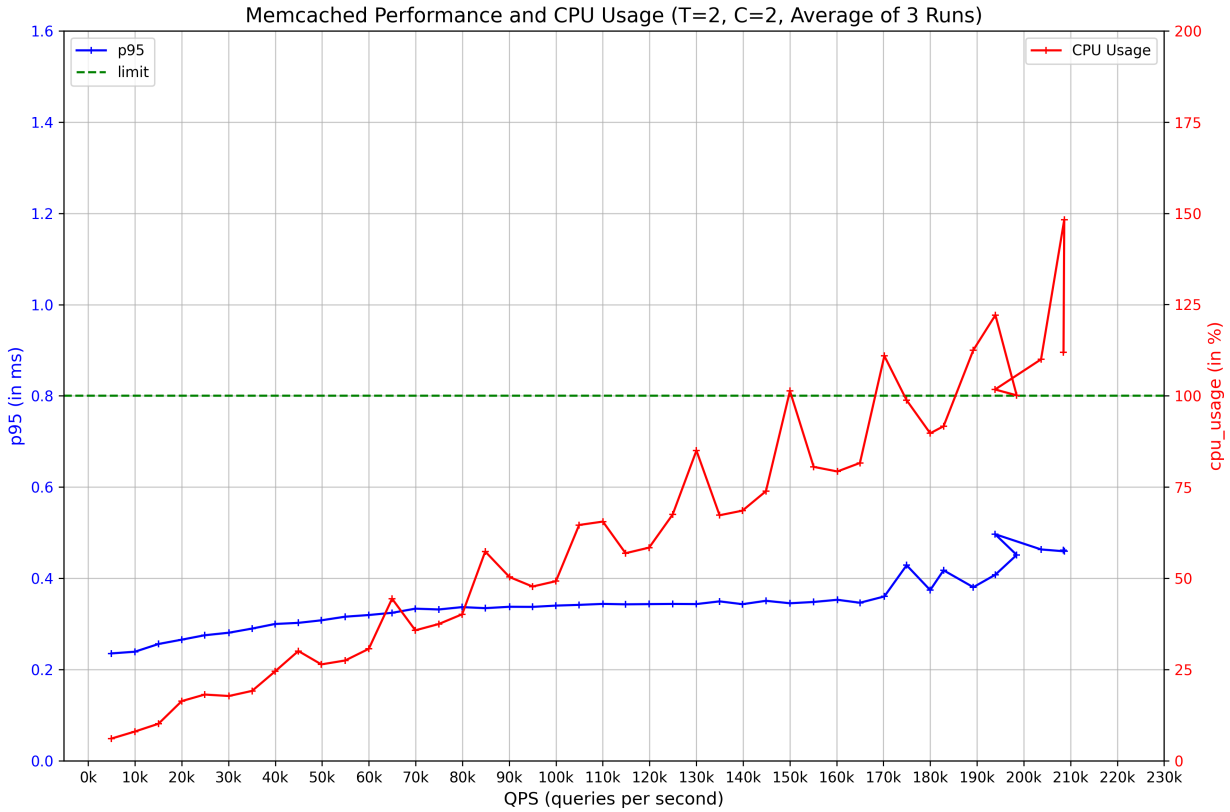

Figure 8: Weighted Average p95 Latency and CPU Utilization vs. Achieved QPS in Memcached (T=2, C=2, Averaged Over 3 Runs)

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: We decided to run freqmine alone as it's the longest job and it presents moderate to high interference with other jobs and scales well with more cores. We run the other jobs in 2 groups concurrently. The decision of which jobs to put in which group was decided as follows: we decided first to put 1 long job per group (ferret and canneal), then we tried to assign the other jobs by looking at the interferences they cause (based on previous questions). Last we switched some jobs because it was experimently better (dedup and radix). More concretly, we first launch canneal, blacksholes and dedup simultaneously, then as soon as they all finish we launch the next 3 jobs ferret, vips and radix simultaneously. Once they terminate, we at last lunch freqmine alone. We use a 60% CPU usage threshold (value determined in figure (7)) to decide whether to allocate 2 cores or only 1 core to memcached.

- How do you decide how many cores to dynamically assign to memcached? Why?
  Answer: We decide on how many cores to assign to memcached based on the CPU usage of the process running it. The reason we use the CPU usage is because of the insight we got from the previous question of this part. Indeed, we observe a strong correlation between the CPU usage and the latency. More concretely, we use a 60% CPU usage threshold to decide whether to allocate 2 cores or only 1 core to memcached. This value comes from the experiments performed in the first question of this part (7), as it is a value that comes a little before the saturation point and respects the SLO. We also experimented with other values and also other techniques (e.g Having 2 thresholds, one to switch from 1 to 2 and a different for 2 to 1), but 60% seemed to be a good stable value that is a somewhat aggressive, but at the same time safe enough to respect the SLO.

- How do you decide how many cores to assign each batch job? Why?
  Answer:
  - blackscholes: 2
  - canneal: 2 or 3
  - dedup: 1
  - ferret: 2 or 3
  - freqmine: 2 or 3
  - radix: 1
  - vips: 1
    We allocate the maximum number of cores (2 or 3, depending on memcached's CPU usage) to all long-running jobs. For shorter jobs that do not represent a performance bottleneck, we assign only a single core to prevent interference with other processes. Although this causes the shorter jobs to take slightly longer, it has no impact on overall performance since they still complete faster than the longer ones. Among

them, only blackscholes runs on two cores, as its execution time is not as short as the others.

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

  Answer:

  - blackscholes: 2
  - canneal: 3
  - dedup: 1
  - ferret: 3
  - freqmine: 3
  - radix: 1
  - vips: 1

    We assign each job a number of threads equal to the maximum number of cores it can utilize. This decision is based on our earlier observation that using more threads than allocated cores does not result in any performance gain.

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

  Answer: canneal, blackscholes and dedup run concurrently, and they are all collocated on at least one core. blackscholes and canneal are collocated on 2.
  ferret, vips and radix run concurrently, and they are all collocated on only one core.
  We run jobs concurrently to achieve an overall better maskespan. We run the freqmine job all by itself because it runs much faster without any interference and with using many processing cores. We run the rest of the jobs three at a time, using all the available cores for each group. We make sure to run long jobs on 2 to 3 cores (depending on memcached cpu usage), while shorter ones on 1 core with exceptions of blackscholes that we run on 2, the idea is to limit the intereference of the smaller jobs on the bottleneck ones. This approach keeps the computer's processors busy and prevents the paired jobs from interfering too much with each other and especially with the longer jobs.

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

  Order (to be sorted): [blackscholes, canneal, dedup], [ferret, vips, radix], [freqmine]
  Why: Running all the jobs sequentially is slow and inefficient, so we execute them in groups. The jobs in each group are executed simultaneously. As soon as a group ends, the subsequent one begins. Hence, to ensure uninterrupted use of the available cores. We run freqmine at the very last alone so that it can use the resources without interference.

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

  Answer: Our policy differs from the one in Part 3 in that it runs most of the jobs in collocation on the same cores.The reason is that we now have significantly fewer compute nodes and cores, and so can not have the luxury of completely avoiding core collocation interferences, or having as much concurrently running jobs. In fact, running all the jobs sequentially has indeed proved to be a very inefficient approach, and also running a lot of them concurrently.

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

Answer: We continuously monitor the CPU usage of memcached using the `cpu_percent()` method from the psutil library. When the CPU usage goes above 60%, memcached uses 2 cores, and the batched jobs are restricted to 2 cores. Otherwise when below 60%, the jobs run on 3 cores and memcached on 1 core. To achieve this, we use `taskset` updates for memcached's process, and the `update` method from the python docker SDK.

- 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: To begin, we created two baseline scheduling strategies to try and beat. The first runs all jobs sequentially, assigning 3 threads per job and allowing each to use all available cores. The second runs all jobs concurrently from the start, again with 3 threads per job and full access to all cores. Our scheduling policy beats both baselines.

  Our second best policy uses the exact same group of jobs as our best, and also runs them in the exact same order. In fact, our best policy was built upon the second-best. The only difference is the assigned threads and cores per job, where we have assigned 3 threads to all the jobs in the second-best policy (except for radix we put 2 threads, because it crashes for an unknown reason with 3). This policy runs ~40s slower than our final policy. Our idea behind changing the thread count per job was to prioritize the bottleneck job in each group of jobs by reducing the interference on them. While this causes the shorter jobs to run longer, they do not become bottlenecks, so it has no impact on the overall makespan. In contrast, the actual bottleneck jobs run faster.

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.

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

Answer:

job name	mean time [s]	std [s]
blackscholes	154.33	11.59
canneal	286.67	16.78
dedup	48.67	0.57
ferret	305.0	14.93
freqmine	274.33	14.84
radix	81.67	8.96
vips	170.33	14.29
total time	884.82	5.32

- **Start time (T_{start}):** The earliest start time among all batch jobs (excluding memcached).
- **End time (T_{end}):** The latest end time among all batch jobs (excluding memcached).
- N_{total}: total number of latency data points recorded during $[T_{start}, T_{end}]$
- $N_{violations}$: number of data points where p95$^{\text{th}}$ latency > 0.8ms.

RUN 1:
The **SLO Violation Ratio** is computed as:

$$\text{SLO Violation Ratio}^{\text{run 1}}_{\text{memcached}} = \frac{N_{\text{violations}}}{N_{\text{total}}} = 0 \tag{2}$$

RUN 2:
The **SLO Violation Ratio** is computed as:

$$\text{SLO Violation Ratio}^{\text{run 2}}_{\text{memcached}} = \frac{N_{\text{violations}}}{N_{\text{total}}} = \frac{1}{84} = 1.2\% \tag{3}$$

RUN 3:
The **SLO Violation Ratio** is computed as:

$$\text{SLO Violation Ratio}^{\text{run 3}}_{\text{memcached}} = \frac{N_{\text{violations}}}{N_{\text{total}}} = \frac{6}{90} = 6.7\% \tag{4}$$

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:

Figure 9: 1A: Memcached 95th percentile latency (left y-axis) and achieved QPS (right y-axis) over time (s), with the SLO threshold (0.8 ms) marked for reference.
1B: Displays the number of CPU cores allocated to Memcached (left y-axis) and achieved QPS (right y-axis) over time (s).
Bottom Plot: Gantt chart indicating the start and end times of benchmark applications. Vertical lines marking job starts and ends may share overlapping timestamps, resulting in color collisions in some cases.

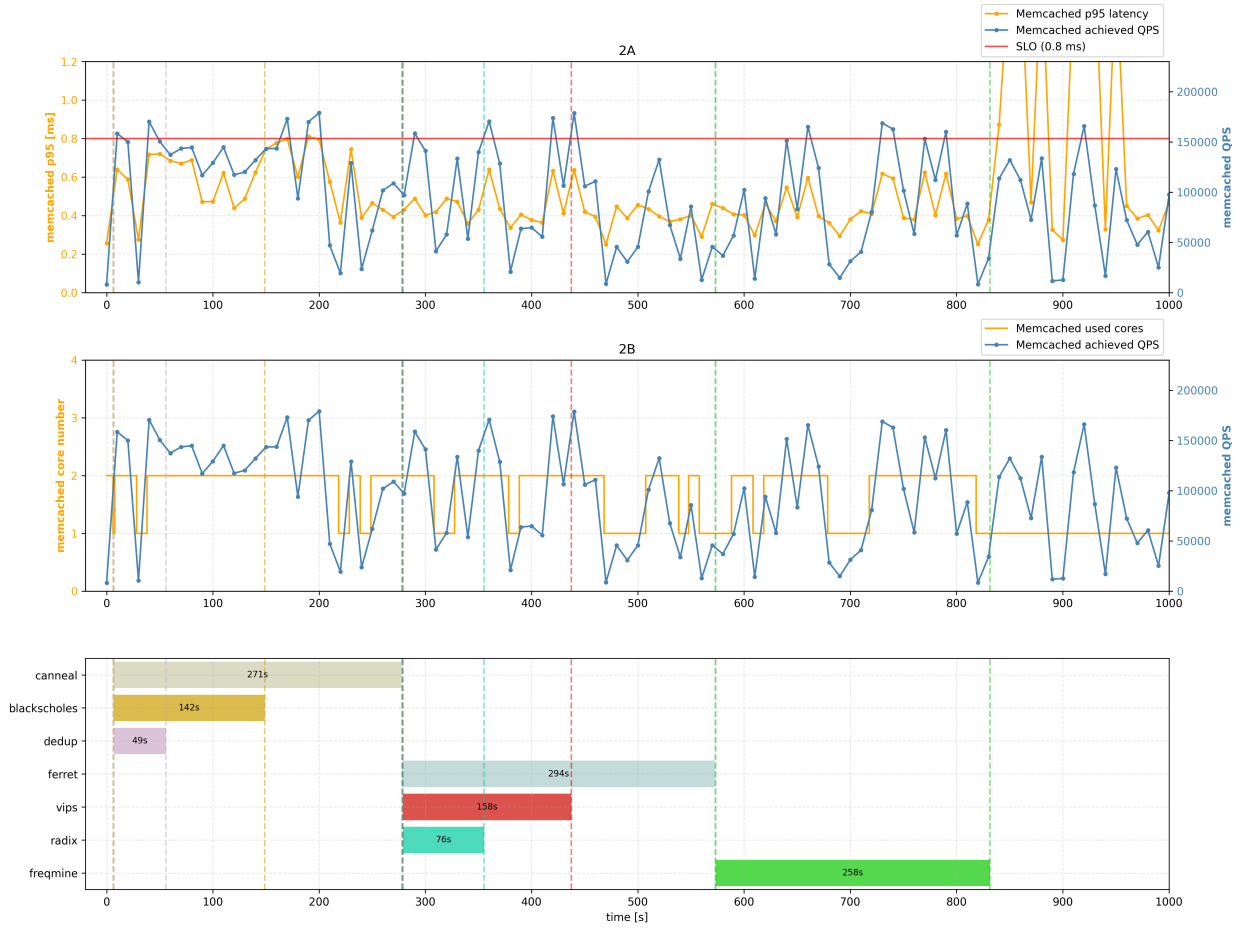

Figure 10: 2A: Memcached 95th percentile latency (left y-axis) and achieved QPS (right y-axis) over time (s), with the SLO threshold (0.8 ms) marked for reference.
2B: Displays the number of CPU cores allocated to Memcached (left y-axis) and achieved QPS (right y-axis) over time (s).
Bottom Plot: Gantt chart indicating the start and end times of benchmark applications. Vertical lines marking job starts and ends may share overlapping timestamps, resulting in color collisions in some cases.

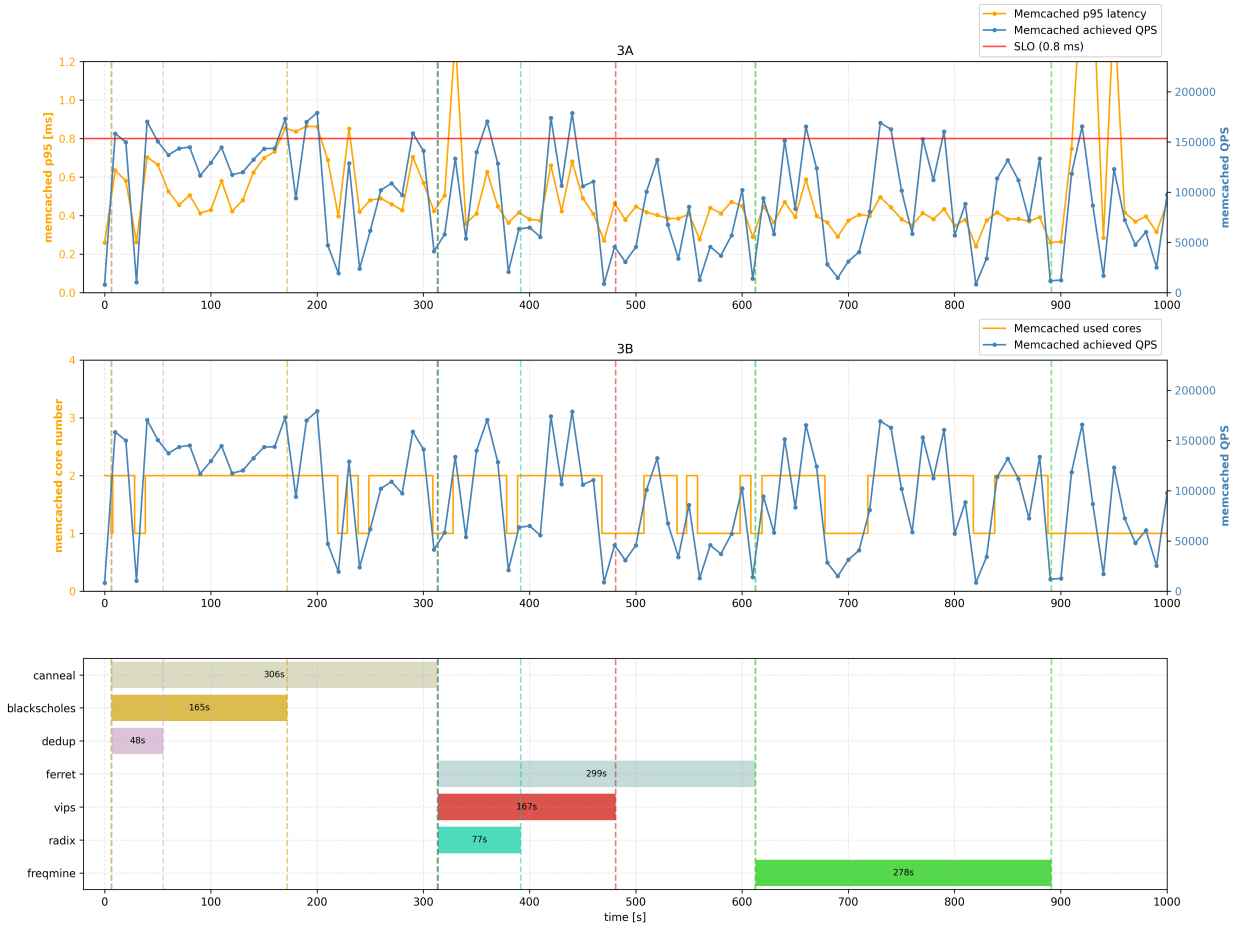

Figure 11: 2A: Memcached 95th percentile latency (left y-axis) and achieved QPS (right y-axis) over time (s), with the SLO threshold (0.8 ms) marked for reference.
2B: Displays the number of CPU cores allocated to Memcached (left y-axis) and achieved QPS (right y-axis) over time (s).
Bottom Plot: Gantt chart indicating the start and end times of benchmark applications. Vertical lines marking job starts and ends may share overlapping timestamps, resulting in color collisions in some cases.

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: Compared to the 10-second interval, the 5-second interval resulted in poorer performance, with a higher average SLO violation rate (5.22% compared to 2.63%). This can be explained by the fact that shorter intervals give the controller less time to analyze the situation and react appropriately by reallocating CPU resources. Our results show that our controller cannot handle the rapidly shifting load of the 5-second QPS interval, resulting in suboptimal performance.

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: The average SLO violation ratio was found to be 5.22%. The first run had 10 violations for 156 datapoints, yielding a 6.41% violation ratio, the second run had a 3.97% ratio with 6 violations for 151 points and the third run had 8 violations for 151 total points, yielding a 5.30% ratio.

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 satisfied that property was 6. Using that, our controller had an average SLO violation rate of 2.7% (respectively 3.2%, 2.45% and 2.45% across our 3 runs).

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

Answer: We determined that 6 was the smallest `qps_interval` that our controller can handle while keeping the SLO violation ratio below 3%. This observation is mostly empirical, based on repeated trials with varying `qps_interval`. Internally, the controller relies on a blocking call to `psutil.cpu_percent(interval=1)`, which introduces a minimum 1 second delay in each loop iteration. Additional delays come from reallocation dynamics that rely on `taskset` and Docker. All these factors and our empirical observations led us to estimate that the controller has a reaction time of a few seconds. If the QPS interval is shorter than this or when the QPS changes too abruptly, the controller does not have enough time to observe the system state and react before the next load change, resulting in higher SLO violation rates.

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	134.83	0.43
canneal	204.18	5.58
dedup	44.32	2.80
ferret	275.16	6.12
freqmine	246.95	7.54
radix	74.56	2.20
vips	144.62	5.66
total time	733.20	13.17

Plots:

Figure 12: 1A: Memcached 95th percentile latency (left y-axis) and achieved QPS (right y-axis) over time (s), with the SLO threshold (0.8 ms) marked for reference.
1B: Displays the number of CPU cores allocated to Memcached (left y-axis) and achieved QPS (right y-axis) over time (s).
Bottom Plot: Gantt chart indicating the start and end times of benchmark applications. Vertical lines marking job starts and ends may share overlapping timestamps, resulting in color collisions in some cases.

Figure 13: 2A: Memcached 95th percentile latency (left y-axis) and achieved QPS (right y-axis) over time (s), with the SLO threshold (0.8 ms) marked for reference.
2B: Displays the number of CPU cores allocated to Memcached (left y-axis) and achieved QPS (right y-axis) over time (s).
Bottom Plot: Gantt chart indicating the start and end times of benchmark applications. Vertical lines marking job starts and ends may share overlapping timestamps, resulting in color collisions in some cases.

Figure 14: 3A: Memcached 95th percentile latency (left y-axis) and achieved QPS (right y-axis) over time (s), with the SLO threshold (0.8 ms) marked for reference.
3B: Displays the number of CPU cores allocated to Memcached (left y-axis) and achieved QPS (right y-axis) over time (s).
Bottom Plot: Gantt chart indicating the start and end times of benchmark applications. Vertical lines marking job starts and ends may share overlapping timestamps, resulting in color collisions in some cases.

<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.