Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Cloud Computing Architecture

Semester project report

Group 050
Kyle Fearne - 22-942-726
Luca Strässle - 17-926-056
Theresa Wakonig - 16-942-633

Systems Group
Department of Computer Science
ETH Zurich
January 18, 2024

Part 3 [34 points]

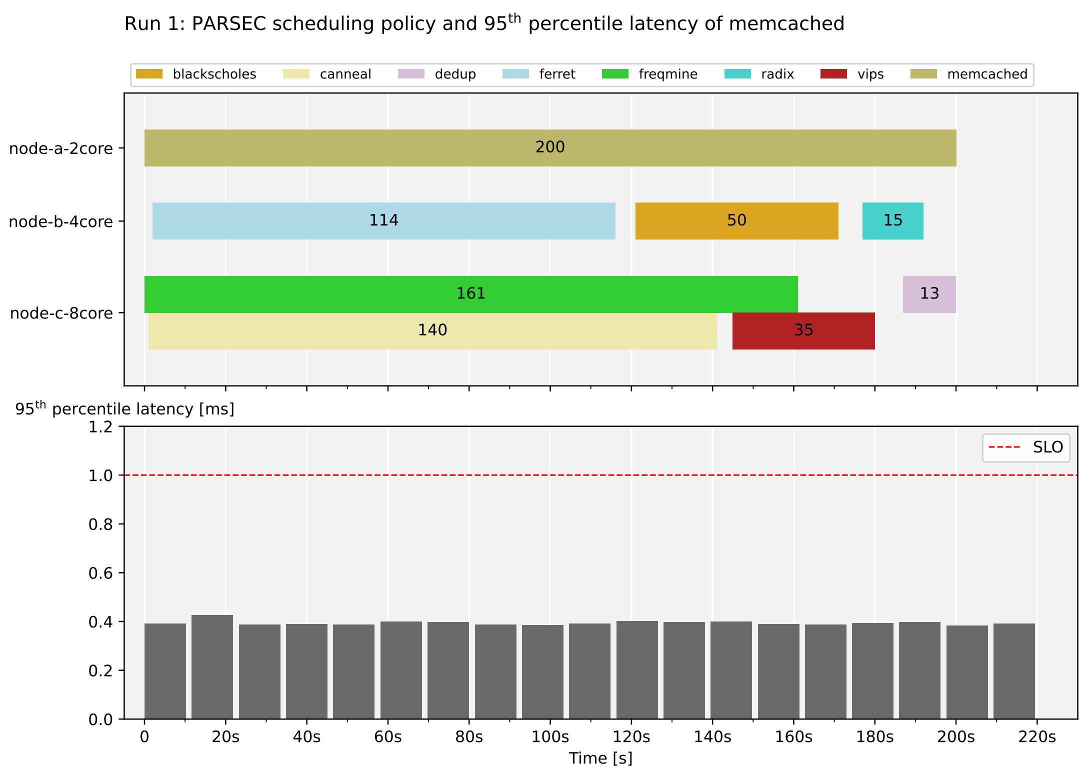
1. **[17 points]** With your scheduling policy, run the entire workflow **3 separate times**. For each run, measure the execution time of each PARSEC job, as well as the latency outputs of memcached running with a steady client load of 30K QPS. For each PARSEC application, compute the mean and standard deviation of the execution time across three runs. Also compute the mean and standard deviation of the total time to complete 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 while the jobs are running.

   Answer:
   With a steady latency value of around 0.4ms our schedule incurrs 0% SLO violation for *memcached*.

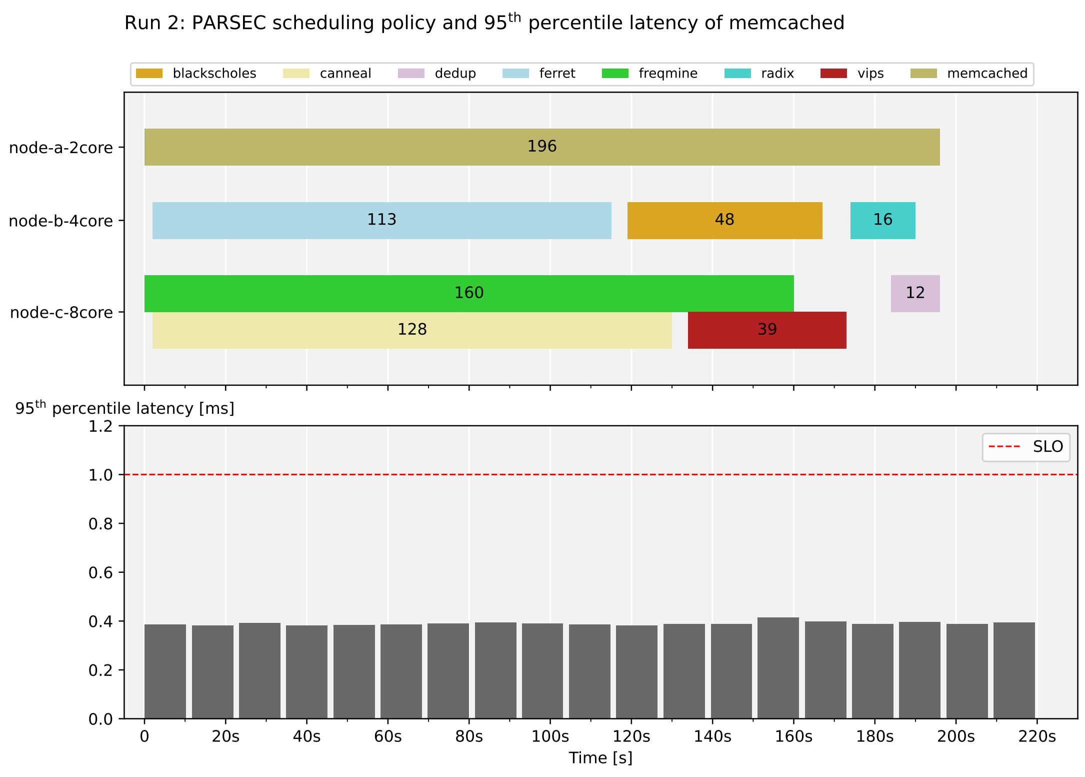
job name	mean time [s]	std [s]
blackscholes	49.33	0.94
canneal	133.67	4.92
dedup	13.33	1.25
ferret	114.0	0.82
freqmine	161.67	1.7
radix	16.0	0.82
vips	38.0	2.16
total time	199.33	2.49

   Create 3 bar plots (one for each run) of memcached p95 latency (y-axis) over time (x-axis) with annotations showing when each PARSEC job started and ended, also indicating the machine they are 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 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.

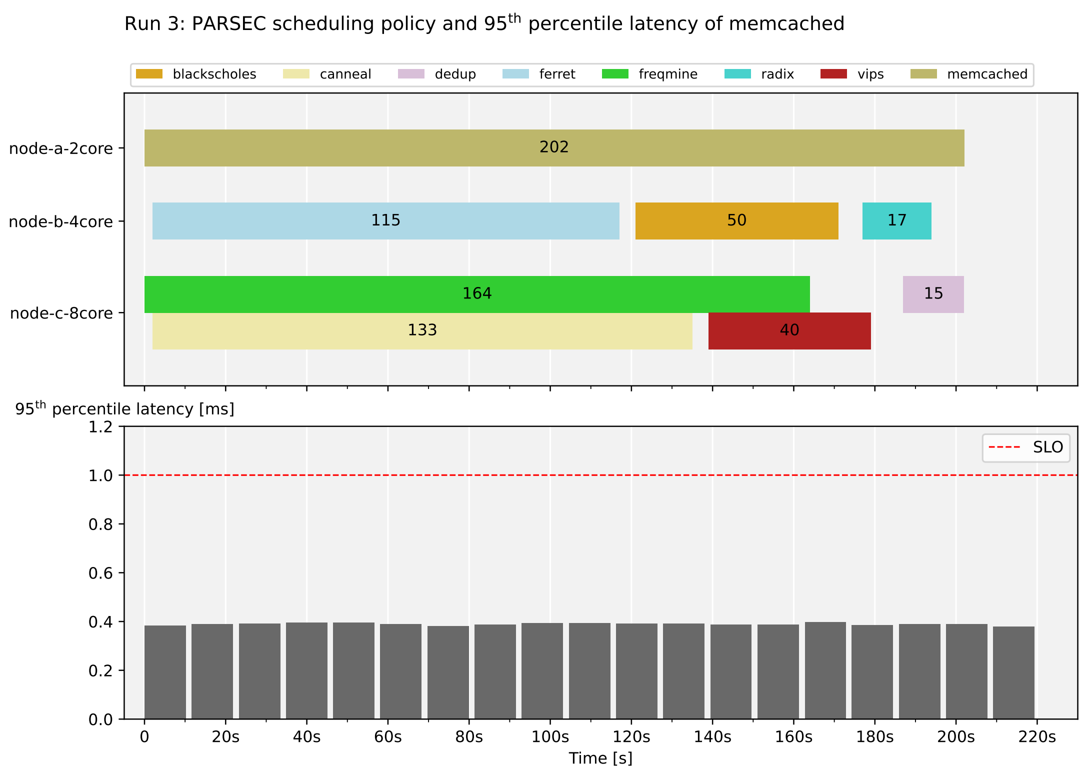
   Plots:



Run 1: PARSEC scheduling policy and 95th percentile latency of memcached
blackscholes
canneal
dedup
ferret
freqmine
radix
vips
memcached
node-a-2core
node-b-4core
node-c-8core
200
114
50
15
161
13
140
35
95th percentile latency [ms]
SLO
Time [s]



Run 2: PARSEC scheduling policy and 95th percentile latency of memcached
blackscholes
canneal
dedup
ferret
freqmine
radix
vips
memcached
node-a-2core
node-b-4core
node-c-8core
196
113
48
16
160
12
128
39
95th percentile latency [ms]
SLO
Time [s]



Run 3: PARSEC scheduling policy and 95th percentile latency of memcached
blackscholes
canneal
dedup
ferret
freqmine
radix
vips
memcached
node-a-2core
node-b-4core
node-c-8core
202
115
50
17
164
15
133
40
95th percentile latency [ms]
1.2
1.0
0.8
0.6
0.4
0.2
0.0
SLO
0
20s
40s
60s
80s
100s
120s
140s
160s
180s
200s
220s
Time [s]

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

- Which node does memcached run on? Why?

  Answer: *Memcached* runs on `node-a-2core` which is a VM of type `t2d-standard-2` featuring two virtual CPUs and 8 GB of memory. Through testing we found that the collocation of *memcached* and PARSEC jobs yields unsatisfactory performance. The PARSEC jobs took more than double the time to finish, or worse, would not even finish executing when run on the same VM as *memcached*. Hence, we decided to run *memcached* in isolation, without any other job running on the same VM. By choosing node *a* for this isolated task we maximize the number of cores available to the other jobs.

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

  Answer: The three most time intensive tasks are `freqmine`, `ferret` and `canneal` and the way they are scheduled has the largest impact on the overall runtime. In Part 2 of the project we saw that `ferret` should not be collocated with other jobs. Since it is a CPU heavy task we scheduled it on all cores on the highcpu node b. At the same time we scheduled `freqmine` and `canneal` on node c. With an experiment we confirmed that scheduling them in parallel gives a better runtime then scheduling them sequentially because they don't scale well to eight cores. For `vips` and `dedup` we witnessed better performance on node c, therefore we placed them there. `blackscholes` and `radix` were scheduled on node b which resulted in nodes b and c finishing at approximately the same time.

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

- Which jobs run concurrently / are colocated? Why?

  Answer: We colocated the following jobs:

  - `freqmine` & `canneal`: With our decision to run all jobs on 4 cores we needed to collocate two of the three long running jobs. As stated above `ferret` should not be scheduled with `freqmine` or `canneal`, which we also confirmed with an experiment. Colocating `freqmine` and `canneal` didn't pose any problems as `canneal` can be colocated well with other tasks (as seen in Task 2).
  - `freqmine` & `vips`: Since `canneal` finishes faster then `freqmine` a second task had to be colocated with `freqmine`. We saw that `vips` runs much faster on node c, is good to colocate with other tasks and takes only a little longer to finish then the remaining part of `freqmine`, so we decided to colocate `freqmine` with `vips`.
  - `vips` & `dedup`: When designing the schedule we wanted `dedup` to run concurrently with the end of `vips`. However in reality this didn't happen, which might be because they are both rather heavy on l1i and llc as seen in Part 2.

The rest of the jobs (`ferret`, `blackscholes`, `radix`) were run exclusively on node *b* (sequentially, without colloacating them) to benefit from the speedup of using all 4 cores simultaneously.

- In which order did you run 7 PARSEC jobs? Why?

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

  Why: In node c our ordering mainly comes from which jobs can be run concurrently which is explained in detail above. In node b we went with a longest job first schedule, however any ordering would have been applicable, since the jobs are running on all cores anyways.

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

  Answer: We did not make use of simultaneous multi-threading (SMT) on our VMs, hence we make the following assumption: 1 *core* = 1 *thread*.
  In Part 2.2 we saw that all jobs except for `dedup` and `canneal` scaled very well to four cores. Except for `radix` none of the jobs scaled very well to eight cores so it made sense to use four cores for `blackscholes`, `ferret`, `freqmine` and `vips`. Since `canneal` is such a long running job and using four cores still results in a speedup over using one or two, it made sense to schedule that job on four cores as well. Otherwise this job alone could have become the bottleneck. `dedup` and `radix` are very short jobs and therefore the number of cores doesn't have a large impact on the overall runtime. In order to make scheduling with the other tasks easier we decided to use four cores as well for `dedup` and `radix`.

  – blackscholes: 4
  – canneal: 4
  – dedup: 4
  – ferret: 4
  – freqmine: 4
  – radix: 4
  – vips: 4

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

  Answer: We modified the `.yaml` files of all the Kubernetes jobs and wrote a simple bash script to launch the jobs.

  Changes in `.yaml` files:

  – node to run the job on (set `cca-project-nodetype` value)
  – number (`num`) of threads per job (add command line argument `-n num`)
  – bind job to specific vCPUs (`id`) (put command line argument `taskset -c id`)
  – resource requests and resource limits to reserve a specific number of vCPUs

  Kubernetes features used: For enforcing the use of a particular number of vCPUs per workload, Kubernetes allows to request and limit resources per container. In our case each job is abstracted as a seprate container. We set the request and limit values for the resource *cpu* in the `.yaml` according to our schedule. For example, to request 4 vCPUs for a job one can add the following lines in the `containers:`

```
resources:
   requests:
     cpu:  "3.5"
   limits:
     cpu:  "4.0"
```

On a VM with 4 cores, the request had to be set to 3.5 because setting it to 4.0 would be to restricitve for the Kubernetes scheduler to still run.

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

  Answer: Overall, our approach was to use the results from Part 2.2 as an approximation for the runtime of each workload at different degrees of parallelization. We then went on and used these values to allocate the jobs taking the longest to execute first, and built the rest of the schedule around it. Additionally, we tried to take the interference patterns from Part 2.1 into account. After performing small collocation tests per node, we designed three schedules and tested them. Based on the results we designed a fourth, improved schedule which performed best and is presented above. The most interesting trade-off was to decide for the three biggest jobs (`freqmine`, `canneal` and `ferret`) whether they should be run at a lower level of parallelization with other jobs running concurrently on the same VM or at a high level of parallelization with exclusive execution of these jobs on the respective node.

Please attach your modified/added YAML files, run scripts, experiment outputs and report as a zip 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 [76 points]

1. [**20 points**] Use the following `mcperf` command to vary QPS from 5K to 125K in order to answer the following questions:

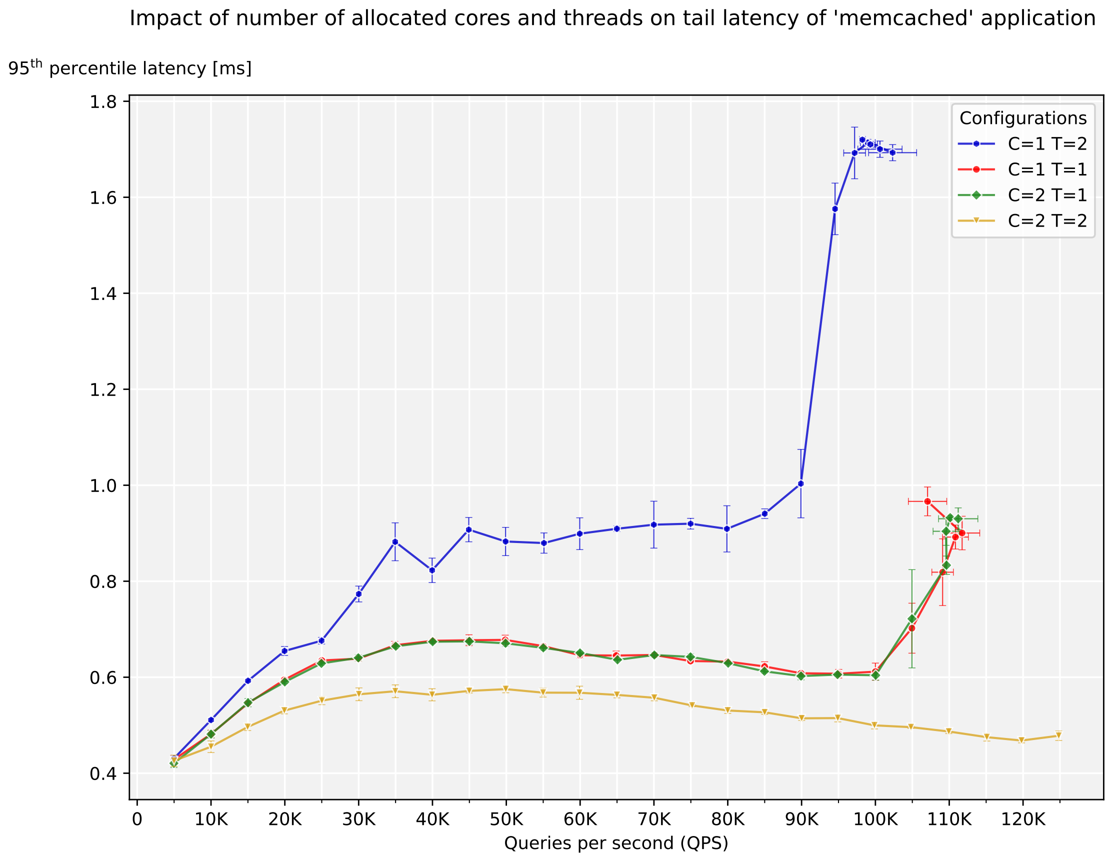
```
$ ./mcperf -s INTERNAL_MEMCACHED_IP --loadonly
$ ./mcperf -s INTERNAL_MEMCACHED_IP -a INTERNAL_AGENT_IP  \
           --noload -T 16 -C 4 -D 4 -Q 1000 -c 4 -t 5 \
           --scan 5000:125000:5000
```

a) [10 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. 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:



The measurements in the above figure were averaged across three runs.

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

Summary:
The figure above shows that the lowest 95th percentile latency was observed when the number of cores and threads was set to two (C=2, T=2). The highest 95th percentile latency was observed when the number of threads was larger than the number of cores allocated (C=1, T=2). It was also evident that when the number of cores was larger than the number of threads, the additional core made no difference - when the number of threads was set to one, the latency was the same whether one or two cores were allocated.

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: The only configuration which can support a load of 125K QPS without violating the 1ms latency SLO is the configuration with two threads and two cores.

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?

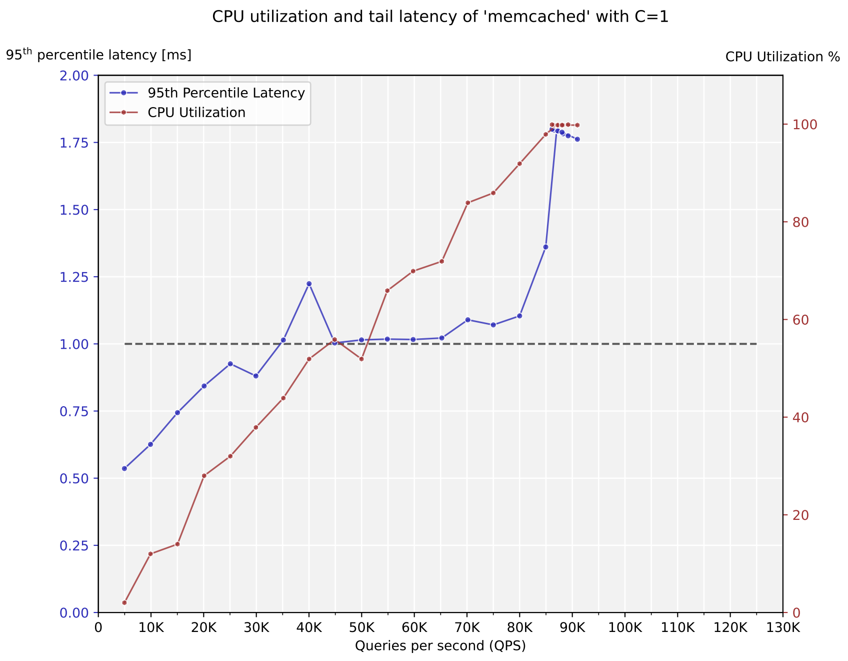
Answer: Two threads are required to guarantee a 95th percentile latency SLO of 1ms. When C=1, the tail latencies are higher, however as these get closer to the 1ms SLO another core can be dynamically added.

d) [7 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 QPS on the x-axis, ranging from 0 to 130K. In each graph, use two y-axes. Plot the 95th percentile latency on the left y-axis. Draw a dotted horizontal line at the 1ms latency SLO. Plot the CPU utilization (ranging from 0% to 100% for $C = 1$ or 200% for $C = 2$) on the right y-axis. For simplicity, we do not require error bars for these plots.

Plots:



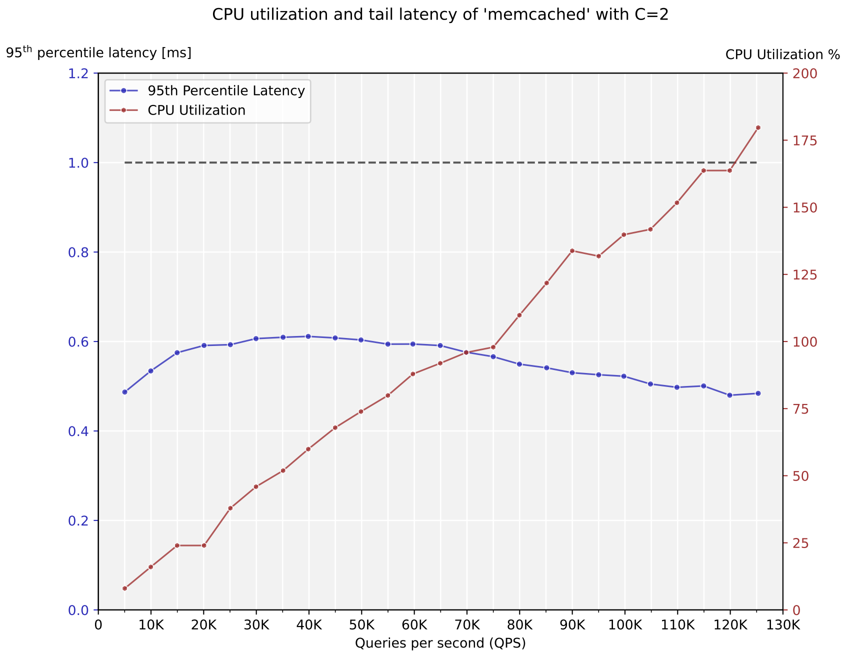


Figure 1: Plots showing variation of CPU utilization and 95th percentile latency based on QPS and number of cores.

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 PARSEC benchmarks on the 4-core VM. The goal of your scheduling policy is to successfully complete all PARSEC 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 PARSEC 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 PARSEC jobs complete successfully and do not crash. Note that PARSEC 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 PARSEC jobs were split into two categories, those assigned one thread and one core and those assigned two threads and two cores. Those in the former category ran on core #1, whilst those in the latter ran on cores #2 and #3. The decision on which job goes into which category was based on the results of Part 2 (effect when collocated with *memcached*) and Part 3 (runtime). *memcached* was assigned two threads, for the reason explained above. It was always run on core #0 and, when required, simultaneously also on core #1. Whenever *memcached*'s CPU utilization threshold for one core was exceeded, the job running on core #1 was paused and *memcached* was assigned this core. Similarly, when the CPU utilization of *memcached* was low, *memcached* was allocated only core #0 and the job previously being executed on core #1 was unpaused.

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

  Answer: Since *memcached* was running on two threads, we had to dynamically assign cores to *memcached* to stay within the 1ms SLO. The plots in Part 4.1 show that at around 35k QPS, (utilization of around 35%), the SLO was exceeded. Therefore, at this utilization threshold *memcached* had to be dynamically assigned another core. Similarly, when running on two cores with low CPU utilization, we dynamically assigned *memcached* just one core so that core #1 could be used by another PARSEC job.

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

  Answer: In our scheduler the number of cores for each PARSEC job is fixed. The ones assigned to category 1 run on one core and can be paused. The ones assigned to category 2 run on two cores and are never interrupted. The assignment of the tasks to the categories was based on runtime, scalability to 2 cores and interference (jobs in the same category will run sequentially and will therefore not interfere). We put `freqmine`, `ferret` and `canneal` which are the longest running tasks into category 2 where they execute on two cores and don't get interrupted. For splitting the rest of the tasks into the categories we made some additional tests to see how long in expectation core #1 is actually available for the PARSEC jobs and how long the three long running tasks are taking. We ended up putting the two short tasks `dedup` and `radix` into category 1. `blackscholes` which has a similar runtime as `vips` but scales worse was put into category 1 and `vips` into category 2.

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

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

  Answer: Each PARSEC job was assigned the same amount of cores as it was assigned threads. This is because having the same number of threads and cores performed best in Part 4.1.

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

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

  Answer: We never assigned one core to multiple jobs. However when *memcached* is only using one core, a category 1 job runs on core #1 at the same time as a category 2 job runs on cores #2 and #3. Additionally *memcached* is always running on one or two cores. Which jobs are running concurrently varies between runs due to the randomness of the *memcached* job. We're presenting what we observed in our three runs where the following jobs were running on the same machine on different cores:

  - `dedup` & `ferret`
  - `radix` & `ferret`
  - `blackscholes` & `ferret`
  - `blackscholes` & `freqmine`
  - `blackscholes` & `canneal`
  - `blackscholes` & `vips`

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

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

  Why: In Part 2 of the project we saw which jobs can and which jobs cannot be run concurrently, due to interference on the same resource. Within each category we tried to order the jobs in a way which minimized the likelihood of two jobs which caused interference on the shared resources from running concurrently. Additionally, we experimented with some different configurations and presented the best result we got here.

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

  Answer: In our policy in Part 3 we also had two categories. We had the jobs running on node b and the tasks running on node c. As in Part 4 one category has double the amount of cores available then the other category. The big difference is that in Part 4 category 1 jobs are sometimes paused for *memcached* to run on two cores. Taking this pause time into account, the long running `ferret`, that ran on node b in Part 3, was put into category 2 in Part 4 and the short running `dedup`, that ran on node c in Part 3, was put into category 1 in Part 4.

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

  Answer: As previously explained, *memcached* was either running on core #0 or on cores #0 and #1, depending on the CPU utilization. The CPU utilization was checked using `psutil` and the number of cores of *memcached* was controlled using the `sudo taskset -a -cp <cores><PID memcached>` command. When switching the number of cores for *memcached* the category 1 job currently running on core #1 had to be paused or unpaused. This was done using the `container.pause()` and `container.unpause()` commands of the Docker Python SDK.

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

  Answer: By not only fixing the number of threads of each job but also the number of cores the scheduler makes sure that all jobs have the same amount of cores and threads available. In Part 4.1 we saw that this is optimal because having more threads then cores increases the tail latency and having more cores then threads leads to idle core(s). Due to the randomness the time that category 1 jobs are paused differs from run to run. The way we split the jobs into the categories now, the category 1 jobs tend to finish at some point where a category 2 job is still running. In order to not have an idle core and to meet the SLO we then assign two cores to *memcached* until the end. We also thought about allowing category 1 to take over unstarted category 2 jobs once finished with its own, and the other way around. However in our experiments the category 1 jobs always finished first and there always was only one unstarted category 2 job left. With the uncertainty due to pausing of jobs running on core #1 we expected only minimal improvement if any for this combination of jobs and decided against implementing it.

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 PARSEC 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 PARSEC job, as well as the latency outputs of memcached. For each PARSEC application, compute the mean and standard deviation of the execution time across three runs. Compute the mean and standard deviation of the total time to complete 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 fractione of the total number of datapoints. You should only report the runtime, excluding time spans during which the container is paused.

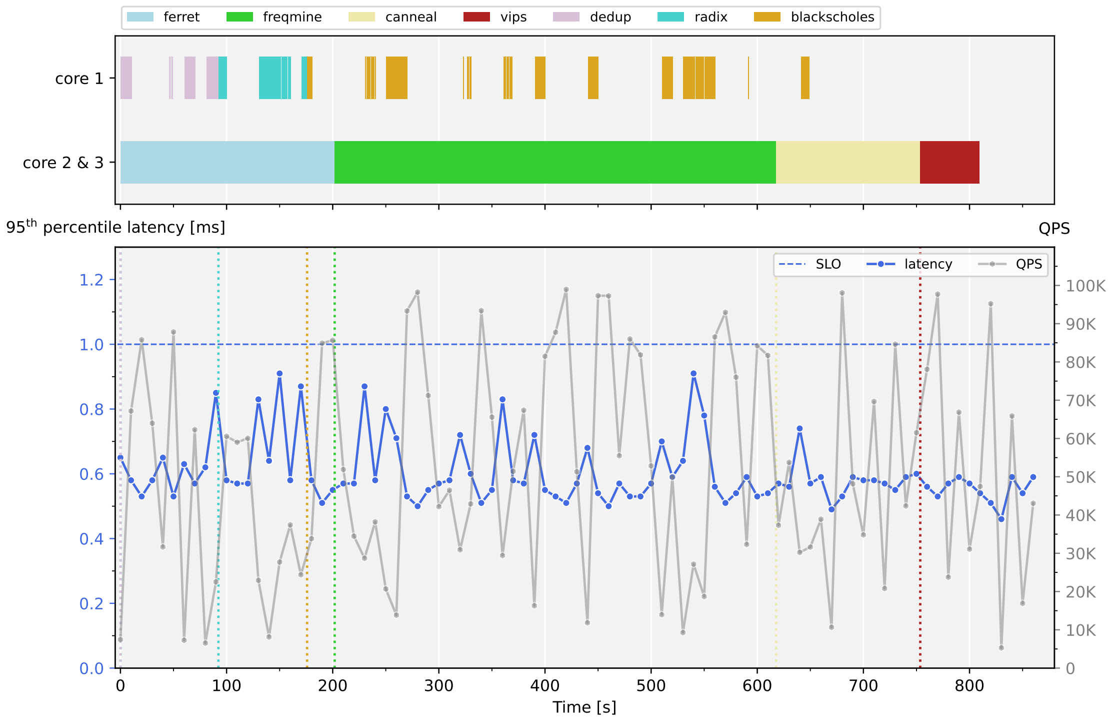
Answer:

job name	mean time [s]	std [s]
blackscholes	105.8	0.23
canneal	137.04	2.96
dedup	33.83	0.31
ferret	199.03	2.24
freqmine	413.82	1.29
radix	41.14	0.28
vips	54.72	0.99
total time	805.89	5.75

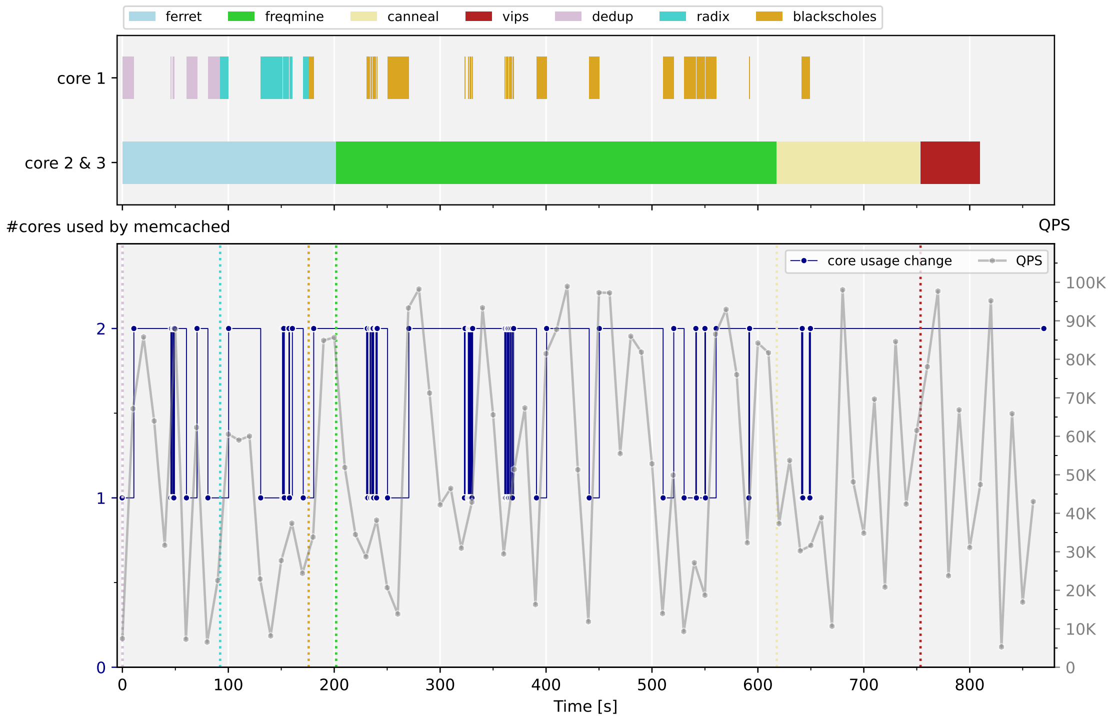
Include six plots – two plots for each of the three runs – with the following information. Label the plots as 1A, 1B, 2A, 2B, 3A, and 3B where the number indicates the run and the letter indicates the type of plot (A or B), which we describe below. In all plots, time will be on the x-axis and you should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

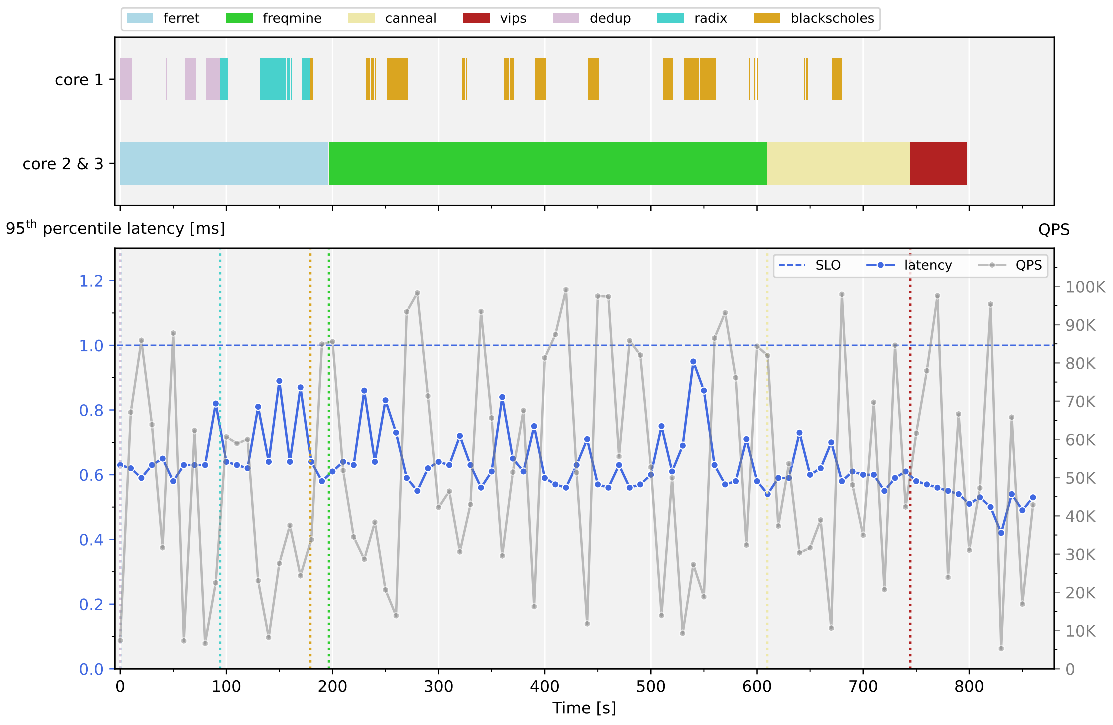
Plot A, run 1: Scheduling policy according to dynamic scheduler and variable load



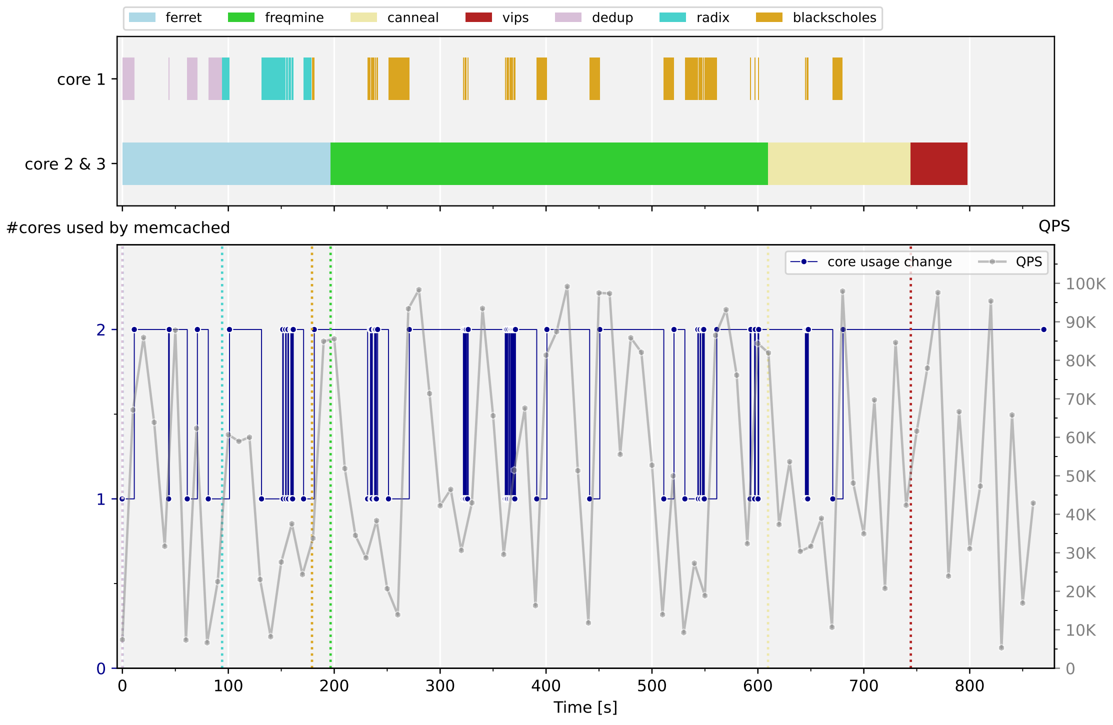
Plot B, run 1: Scheduling policy according to dynamic scheduler and variable load



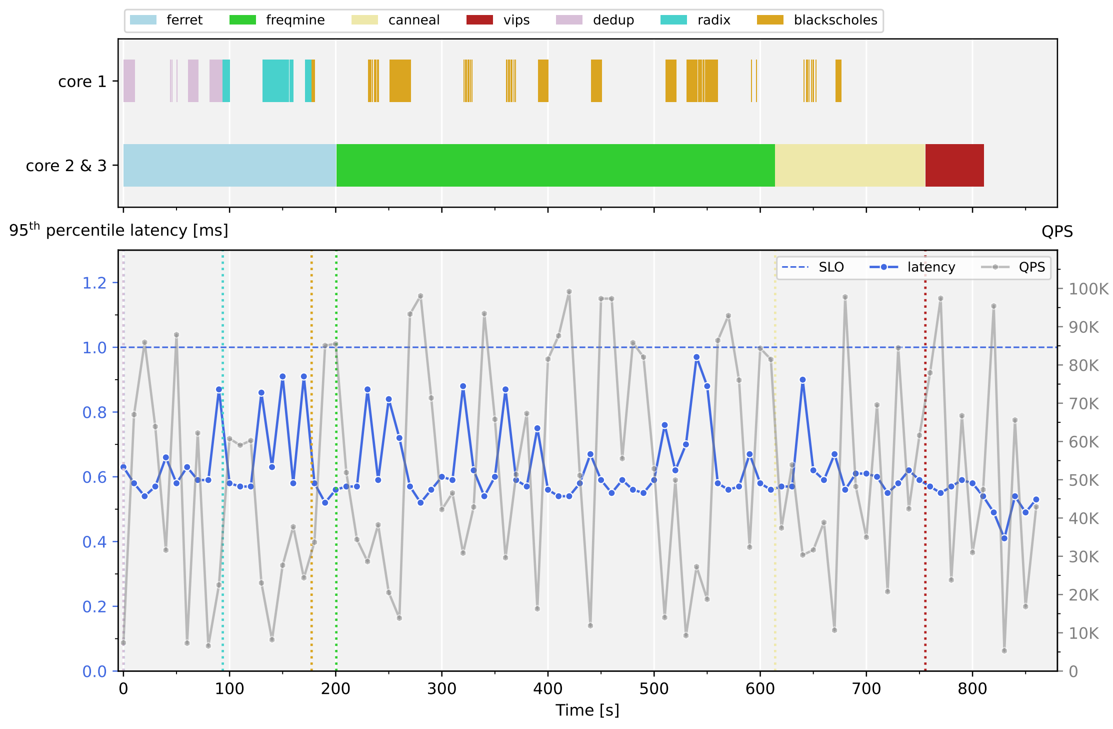
Plot A, run 2: Scheduling policy according to dynamic scheduler and variable load



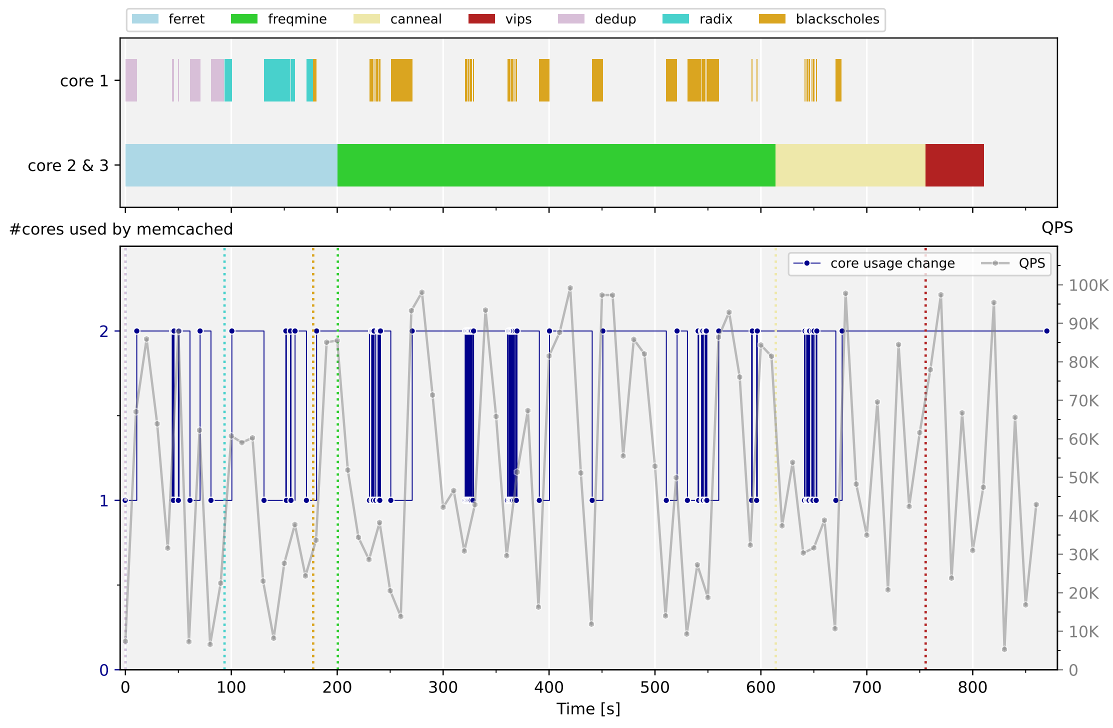
Plot B, run 2: Scheduling policy according to dynamic scheduler and variable load



Plot A, run 3: Scheduling policy according to dynamic scheduler and variable load



Plot B, run 3: Scheduling policy according to dynamic scheduler and variable load



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: Our policy performs very similarly with this interval as there are still no SLO violations. This is owed to the fact that CPU usage is polled every 0.5s and the threshold we set for switching between one and two cores was fine tuned. Higher load variability results in more switches in cores allocated to *memcached*, impacting the category 1 jobs.

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?

Answer: 0%. With our controller, we never exceed the 1ms SLO with the 5-second time interval trace.

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: An interval of 2.75 seconds yielded an SLO violation ratio of approximately 2.6%.

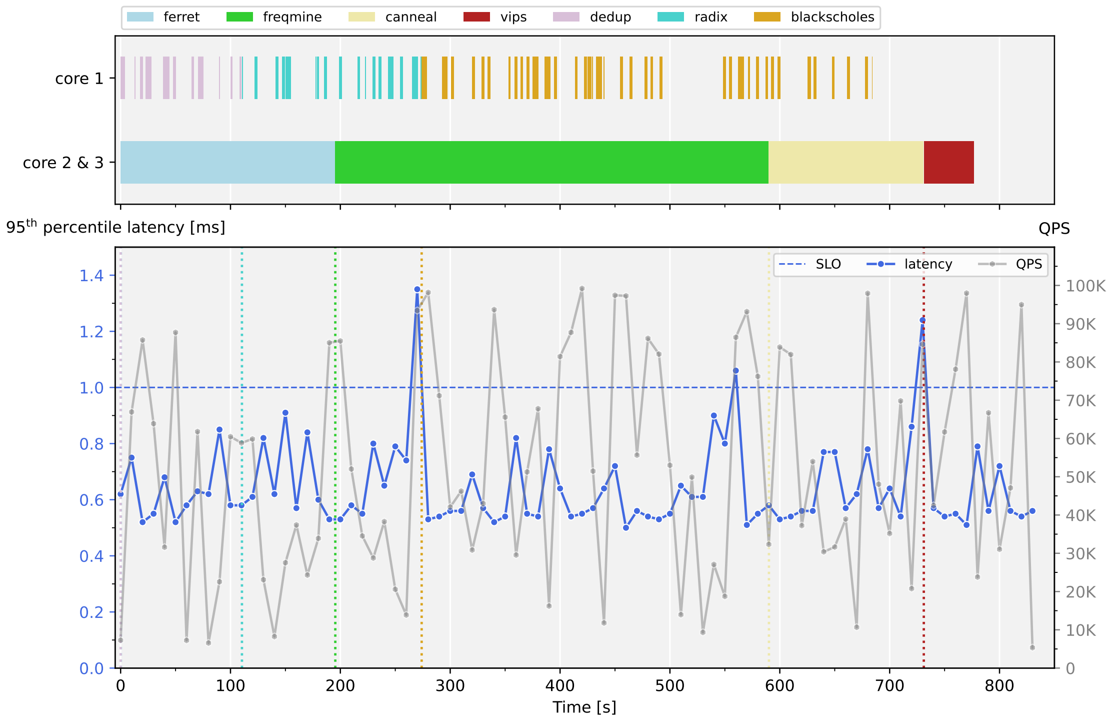
What is the reasoning behind this specific value? Explain which features of your controller affect the smallest `qps_interval` interval you proposed.

Answer: With a lower qps_interval our controller will pause and unpause containers more frequently. Additionally the frequency at which our script polls for CPU utilization (2 times/second) has an affect on the smallest qps_interval value possible. The process of finding this value was experimental. As previously mentioned in Part 4.4, the SLO is never violated with an interval of 5 seconds. Initially, an interval of 2.5 seconds was tested but this yielded a violation ratio of $\approx$ 3.9%. An interval of 3 seconds yielded a violation ratio of $\approx$ 1.7%. By taking a value in between, the interval of 2.75 seconds brought us as close as possible to this 3% SLO violation ratio - with a violation ratio of $\approx$ 2.6%

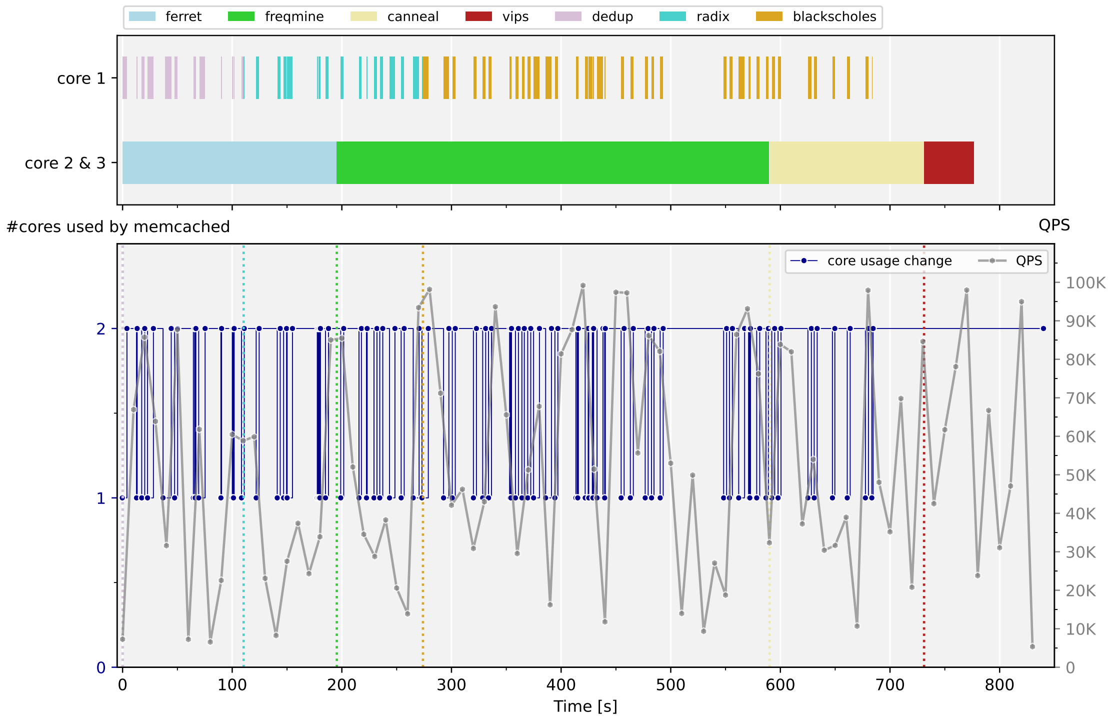
Use this `qps_interval` in the command above and collect results for three runs. Include the same types of plots (1A, 1B, 2A, 2B, 3A, 3B) and table as in Question 3.

Plots:

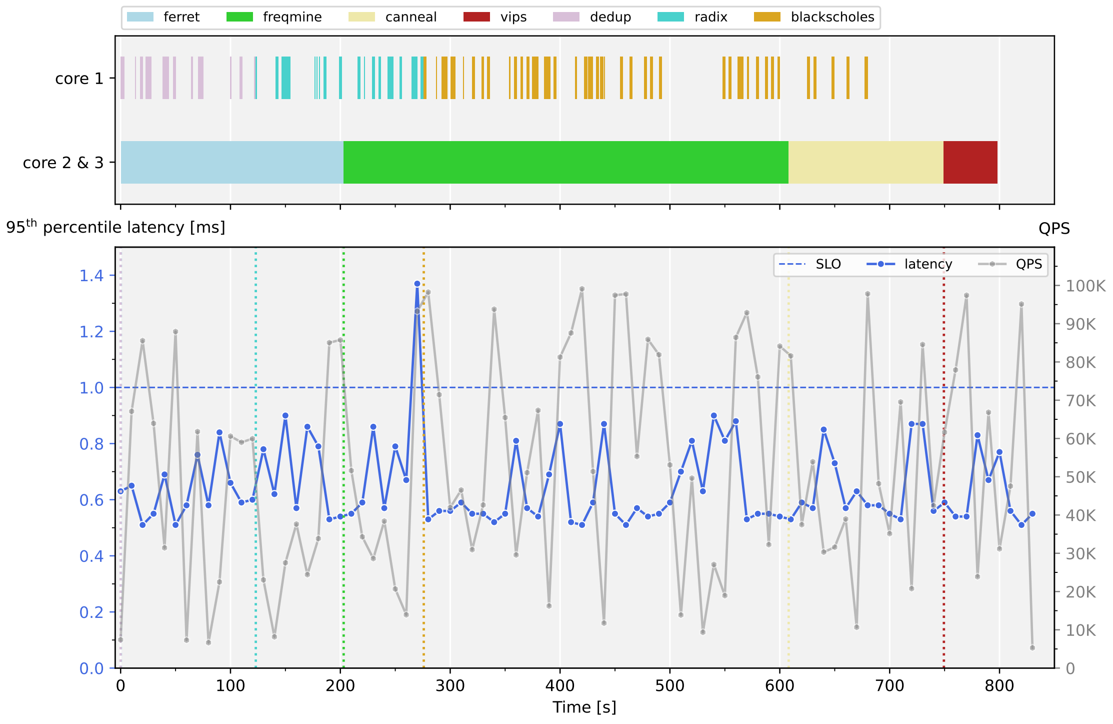
Plot A, run 1: Scheduling policy according to dynamic scheduler and variable load



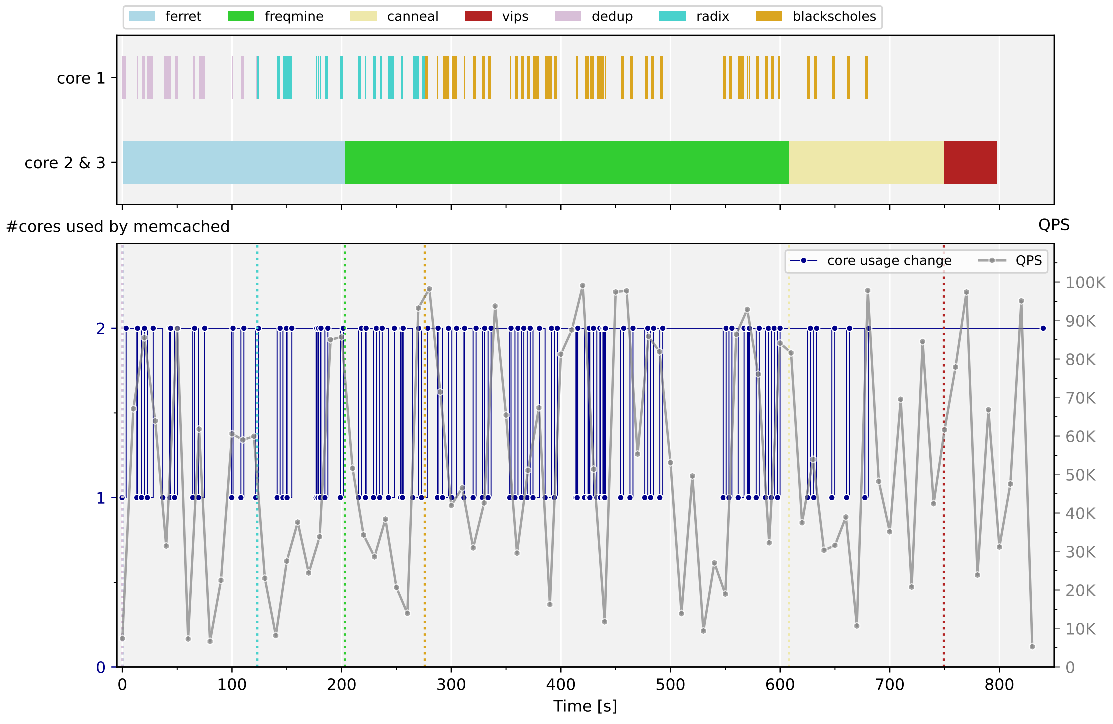
Plot B, run 1: Scheduling policy according to dynamic scheduler and variable load



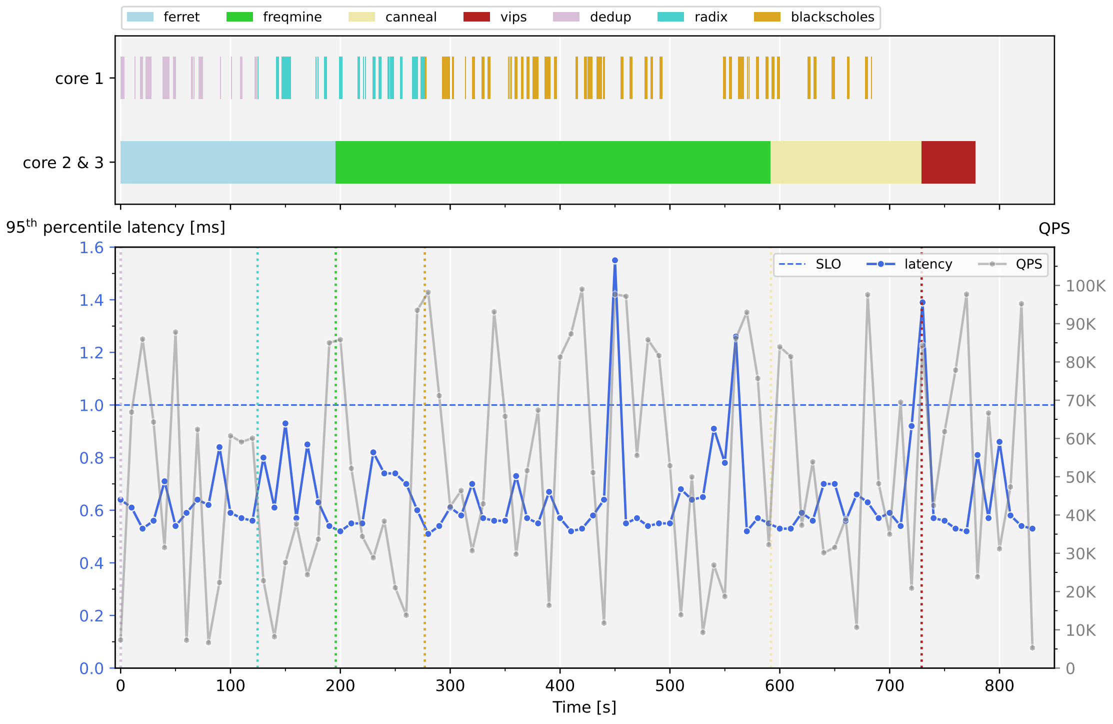
Plot A, run 2: Scheduling policy according to dynamic scheduler and variable load



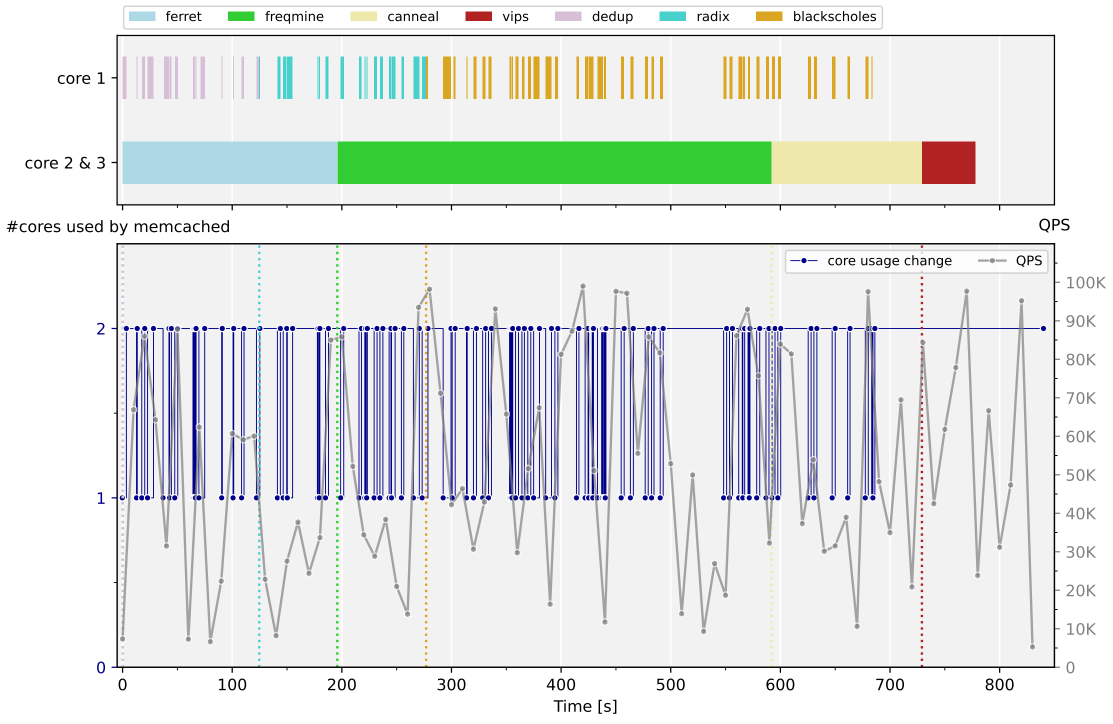
Plot B, run 2: Scheduling policy according to dynamic scheduler and variable load



Plot A, run 3: Scheduling policy according to dynamic scheduler and variable load



Plot B, run 3: Scheduling policy according to dynamic scheduler and variable load



job name	mean time [s]	std [s]
blackscholes	105.76	0.89
canneal	139.52	1.87
dedup	31.19	0.26
ferret	197.5	3.46
freqmine	398.31	4.57
radix	41.82	0.35
vips	47.67	1.53
total time	784.16	9.87