Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Systems@ETH Zürich

Cloud Computing Architecture

Semester project report

Group 029
David Feng - 22-911-143
Timon Fercho - 19-942-374
Mert Ertugrul - 21-960-795

Systems Group
Department of Computer Science
ETH Zurich
May 24, 2023

- **Please do not modify the template!** Except for adding your solutions in the labeled places, and imputing 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 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 [1] 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: We do not violate the SLO at any data point. Therefore, our SLO violation ratio is 0.

   Create 3 bar plots (one for each run) of memcached 95th percentile 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 95th percentile 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. .

job name	mean time [s]	std [s]
blackscholes	71.33	0.47
canneal	167.33	5.56
dedup	48.67	2.49
ferret	110.67	0.47
freqmine	124.67	0.47
radix	9.67	0.94
vips	43.67	5.91
total time	169.33	4.50

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

   - Which node does memcached run on? Why?

     Answer: We chose to put memcached on a dedicated core on node A since we observed SLO violations when it was colocated on the same core with another job, likely due to memcached sensitivity to CPU and L1 cache interference we observed in part 1. We also noticed that memcached does not require the compute power of cores B and C to meet the SLO, so allocating it to core A made the most sense. This approach also leaves room for the other jobs to take advantage of the higher compute power available on nodes B and C.

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

     Answer:

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

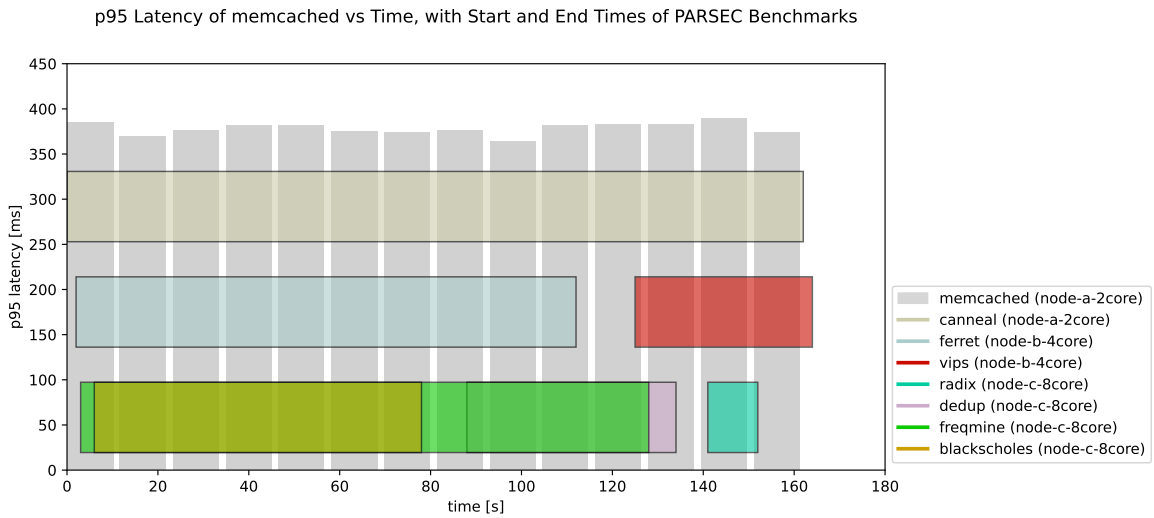

Figure 1: 95th percentile latency of Memcached over time and scheduled PARSEC jobs (Run 1)

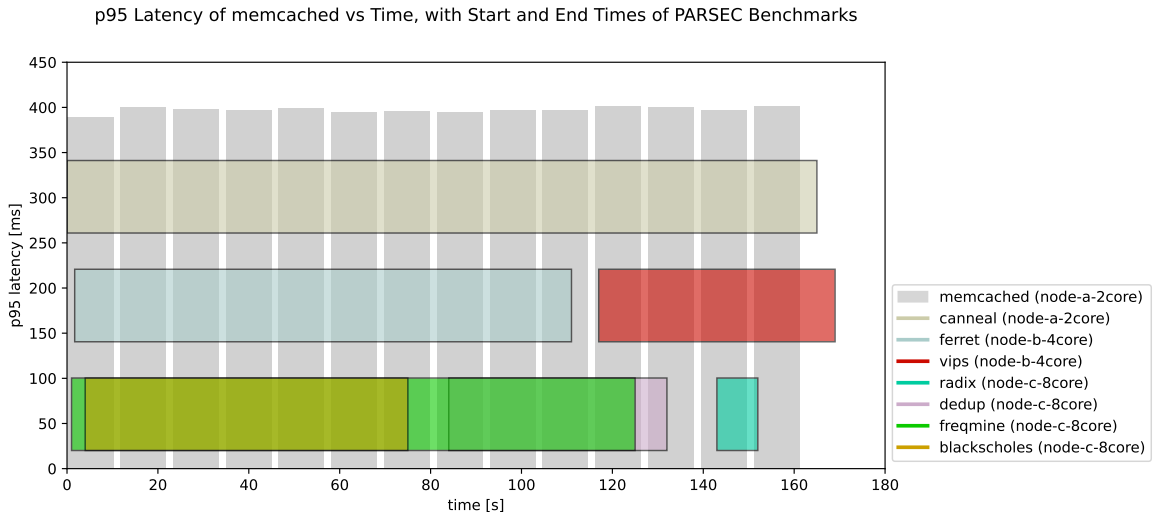

Figure 2: 95th percentile latency of Memcached over time and scheduled PARSEC jobs (Run 2)

- blackscholes: Based on the light interference behavior we observed in part 2 and the great scaling behavior, we decided to run blackscholes on node C sharing all 8 cores with freqmine. The interference only lead to a 30 percent increase in runtime for freqmine (130s vs 100s) but allowed us to complete blackscholes reasonably quickly because it is parallelized over all 8 cores. Moreover, it was shown in part 2a that blackscholes scales linearly with the number of threads used, so allocating more cores to blackscholes improves its run time.
- canneal: From part 2, we noticed that canneal alongside with radix and blackscholes had the lightest general interference behavior. We also concluded from part 2a that canneal and memcached can be safely colocated because canneal has low interference with cpu and l1i, which memcached utilizes. Although canneal, alongside freqmine and ferret, belongs to the jobs with the longest overall runtime, we could effectively

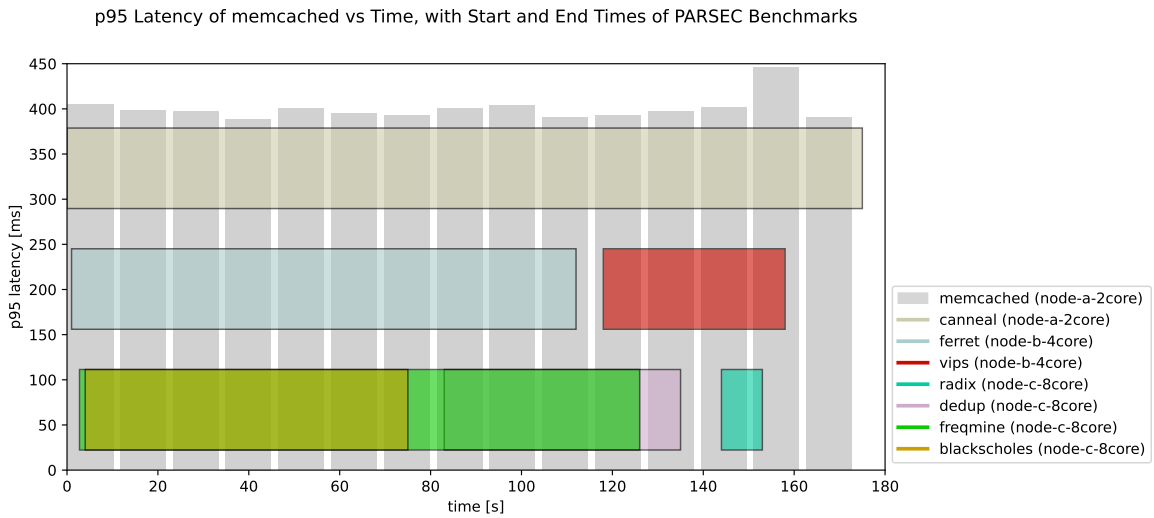

Figure 3: 95th percentile latency of Memcached over time and scheduled PARSEC jobs (Run 3)

make use of the spare core on node A where memcached was running without violating SLO. Despite the few resources (1 core) being allocated to canneal, the run time nicely aligns with the completion time of all other jobs on node B and C, giving them more resources.

- dedup: Similarly to blackscholes, we run it alongside freqmine on node C. We also observed low interference with freqmine and schedule it once blackscholes completes. This approach allowed us to complete all three jobs on node C within less than 140s.
- ferret: From part 2 we know that ferret is the most demanding job on the a node's resources. Therefore, we schedule it on all 4 cores of node B on its own. Despite its medium to high overall execution time, it executes reasonably fast within 120s and does not cause any interference with other jobs or memcached.
- freqmine: As discussed above, freqmine belongs to the longest-running jobs, which is why we decided to run it on node C with all 8 cores. Luckily, it does not cause much interference, so we could complete all of blackscholes and most of dedup during its execution time.
- radix: Radix is a fairly short-running and low-interference job. We used the available resources on node C after completion of all other jobs there, to quickly finish the job.
- vips: We observed medium to high interference and overall runtime for vips in the previous parts, so we decided to schedule it on its own on node B once ferret has been completed, giving vips access to all 8 cores in the VM.

- Which jobs run concurrently / are colocated? Why?

  Answer: As discussed above, we ran long-running and low-interference freqmine on node C with 8 cores and colocated it with short-running and low/medium interference blackscholes and dedup. This way, we could complete jobs that required more time while also granting some resources to jobs that are less time-intensive. We did not observe a high interference penalty between the jobs and also did not violate SLO because we ran Memcached on node A with a dedicated core. As a result, we could make more effective

use of the high compute resources available on node C.

- In which order did you run 7 PARSEC jobs? Why?
  Order (to be sorted): canneal, ferret, freqmine, blackscholes, dedup, vips, radix
  Why: The order is a consequence of our design choices mentioned above. We try to schedule long-running jobs early, and we then aim to minimize and balance the overall execution time across all nodes by splitting short-running jobs across each node. Part 2 gave us insight into general execution times for each job, which informed our scheduling decisions for part 3.
- How many threads have you used for each of the 7 PARSEC jobs? Why?
  Answer: We gave each job as many threads as we allocated cores to avoid scheduling overheads (context switching) when using more threads than cores and also utilizing the available compute resources as best as possible. Although dedup does not scale as well to 8 cores/8 threads, we still preferred it over scheduling ferret on node C due to ferret's bad interference behavior and would probably not have improved the overall runtime as node C completed first in our schedule anyways. However, according to our results in part 2, all other jobs scaled linearly with the number of threads, so giving each job the maximum number of available treads maximized the overall efficiency of the schedule, with the exception of dedup.
  - blackscholes: 4
  - canneal: 1
  - dedup: 8
  - ferret: 4
  - freqmine: 8
  - radix: 8
  - vips: 4
- Which files did you modify or add and in what way? Which Kubernetes features did you use?
  Answer: First, we defined the temporal dependencies and parallelism of our schedule using a custom json file. This json file had an object for each node, and each object contained the node's name and information about the jobs that would be run on that node, including order and colocation information. We used this json file in our scheduler to launch the jobs based on each job's yaml configuration file. Within the config, we specified the node to run on ("nodeSelector"), the number of threads ("./run ... -n X"), and assigned the core(s) to run on ("taskset -c X ...").
- Describe the design choices, ideas and trade-offs you took into account while creating your scheduler (if not already mentioned above):
  Answer: At first, we tried to create the perfect schedule, trying to take into account the scalability of jobs, the projected run time of jobs, the resource requirements, and the potential interference between jobs. However, we eventually realized that we cannot prioritize every factor, so we had to make some trade-offs. For example, we tried to schedule node C to have no interference between pairs of colocated jobs, but we could only do our best to minimize it, picking jobs that had minimal interference and keeping

the interference period as short as possible. Another instance of this trade-off is dedup, which runs optimally on 4 cores. Yet we decided to run it on 8 cores on node C, which, in the broader picture, was a trivial consideration because node C finished before either of the other two nodes.

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:

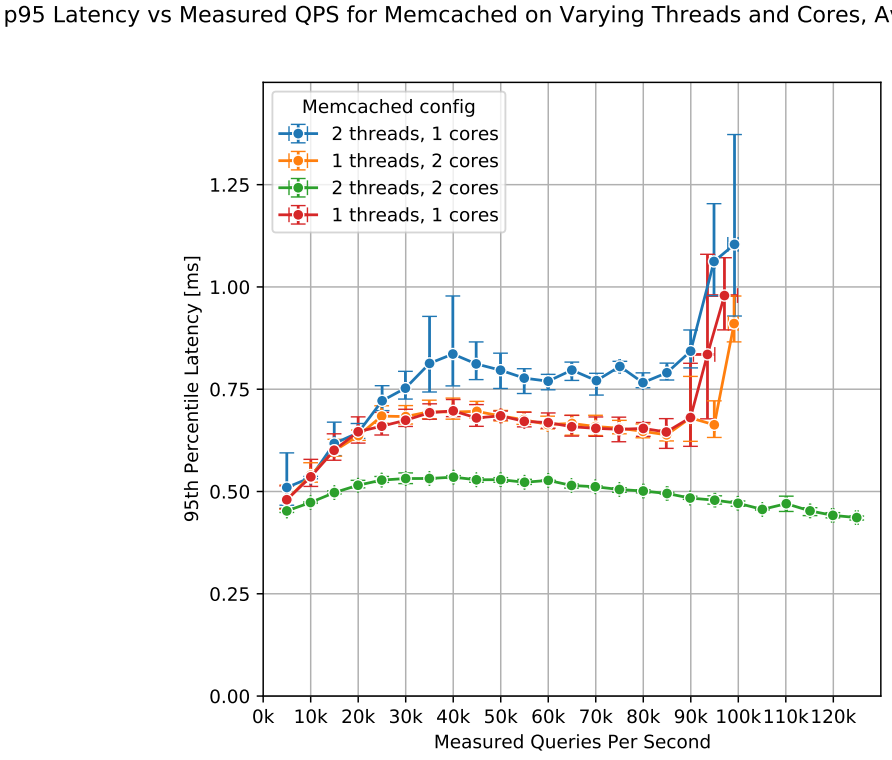

Figure 4: Graph of 95th percentile latency of memcached with varying threads and cores as a function of measured queries per second, averaged across 3 trials

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 configuration that performs the worst is the case with 2 threads and 1 core, which makes sense because context switching – a very expensive operation – is required, increasing the latency. The cases of 1 thread and 1 core and 1 thread and 2 cores performs similarly because a single thread cannot take advantage of 2 cores. The best-performing configuration is 2 threads and 2 cores, which allows memcached to take advantage of multi-threading without the performance penalty of context switching.

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: We need 2 threads and 2 CPU cores. All the other configurations cannot even reach a measured performance of 125K QPS and are bound to around 100K 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: We should use 2 threads when dynamically varying the number of cores allocated to memcached. This is because scenarios with 1 thread would be unable to run the desired higher loads greater than 100K QPS, and more cores can always be allocated for when higher loads are encountered.

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:

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
```

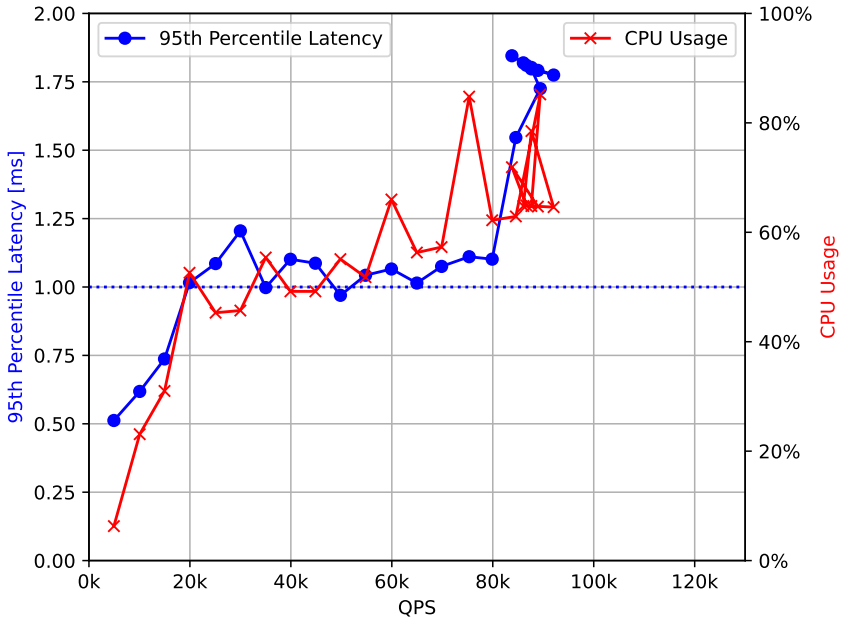

Figure 5: Graph of 95th percentile latency of memcached CPU utilization for 1 core.

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:

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

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

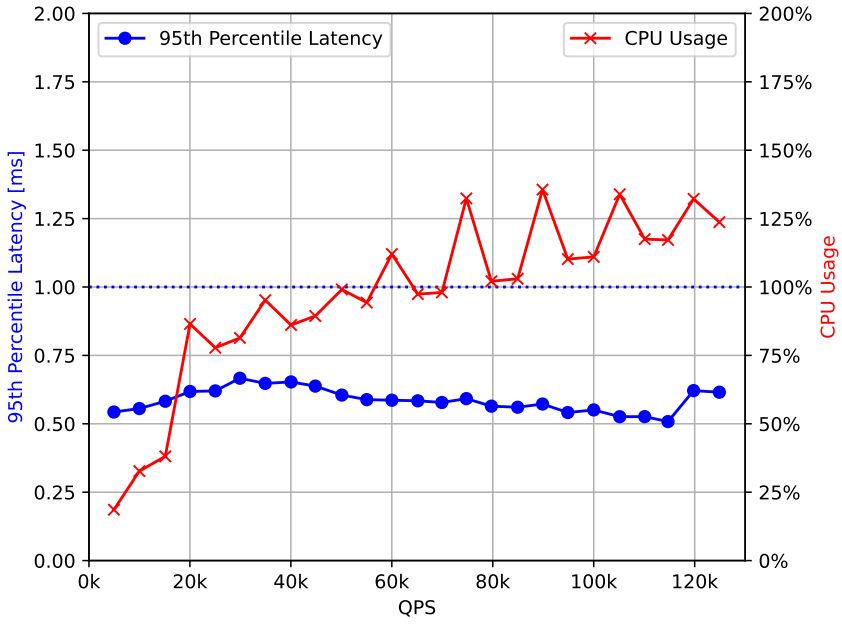

Figure 6: Graph of 95th percentile latency of memcached CPU utilization for 2 cores.

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

- How many threads do you use for each of the PARSEC job? Why?
  Answer:
  - blackscholes:
  - canneal:
  - dedup:
  - ferret:
  - freqmine:
  - radix:
  - vips:

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

- In which order did you run the PARSEC jobs? Why?
  Order (to be sorted): blackscholes, canneal, dedup, ferret, freqmine, radix, vips

Why:

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

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

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

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 fraction 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		
canneal		
dedup		
ferret		
freqmine		
radix		
vips		
total time		

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:

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:

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:

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:

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

Answer:

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:

job name	mean time [s]	std [s]
blackscholes		
canneal		
dedup		
ferret		
freqmine		
radix		
vips		
total time		