Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Cloud Computing Architecture

Semester project report

Systems Group
Department of Computer Science
ETH Zurich

Instructions

- Please do not modify the template, except for putting your solutions, group number, names and legi-NR.

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 datapoints with 95th percentile latency > 2ms, as a fraction of the total number of datapoints. Do three plots (one for each run) of memcached p95 latency (y-axis) over time (x-axis) with annotations showing when each parsec job started.

   For Part 3, you must use the following `mcperf` command:

```
$ ./mcperf -s MEMCACHED_IP --loadonly
$ ./mcperf -s MEMCACHED_IP -a INTERNAL_AGENT_A_IP -a INTERNAL_AGENT_B_IP  \
           --noload -T 6 -C 4 -D 4 -Q 1000 -c 4 -t 20 \
           --scan 30000:30500:10
```

job name	mean time [s]	std [s]
dedup	22.33	0.94
blackscholes	80.33	0.47
ferret	287.33	0.94
freqmine	71.67	0.47
canneal	161.33	1.25
fft	80.00	4.90
total time	325	1.41

Run	SLO violation ratio (%)
Run 1	0 %
Run 2	0 %
Run 3	0 %

The results of the three runs can be observed in the following figures:

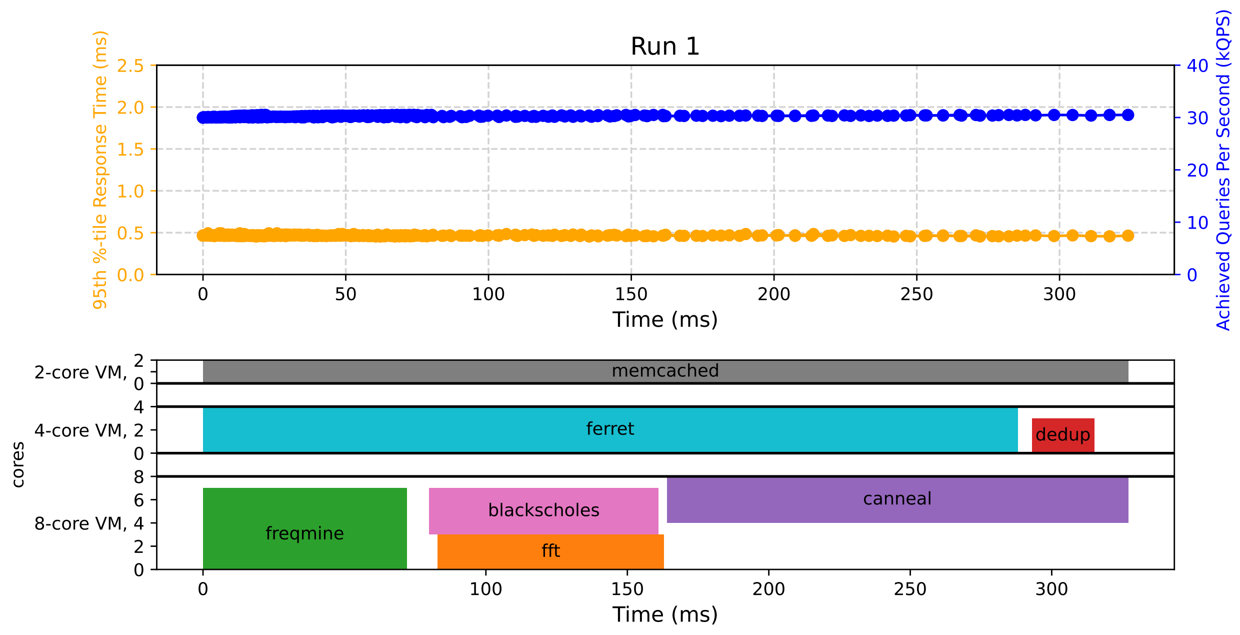


Figure 1: Run 1 of static scheduler

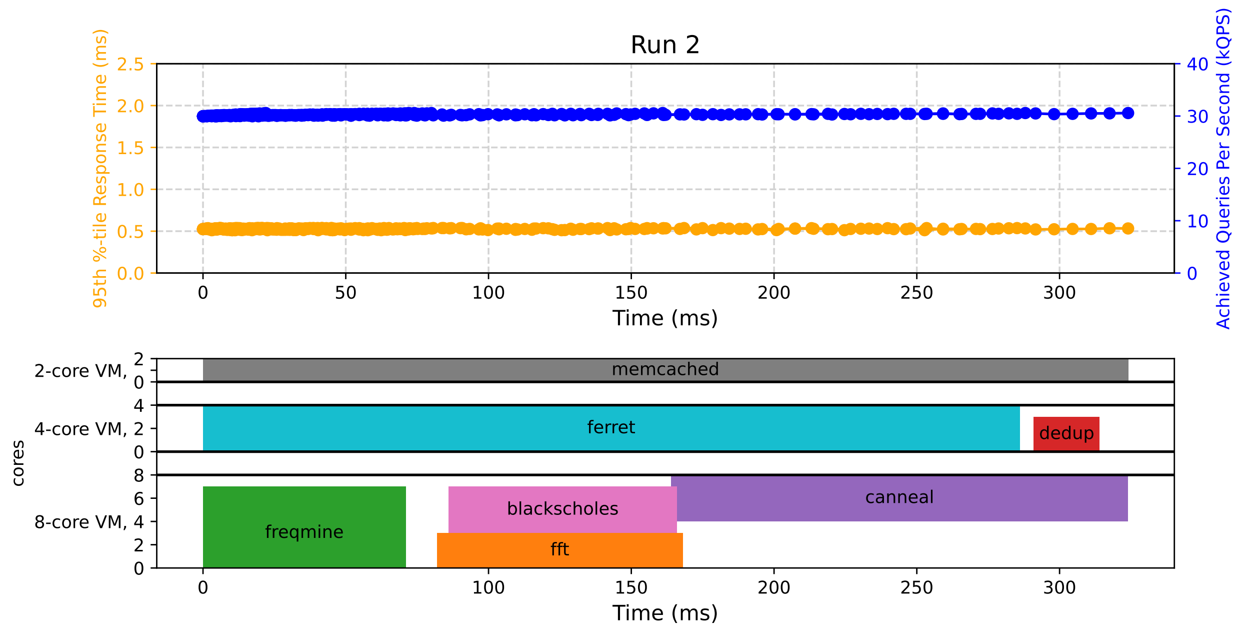


Figure 2: Run 2 of static scheduler

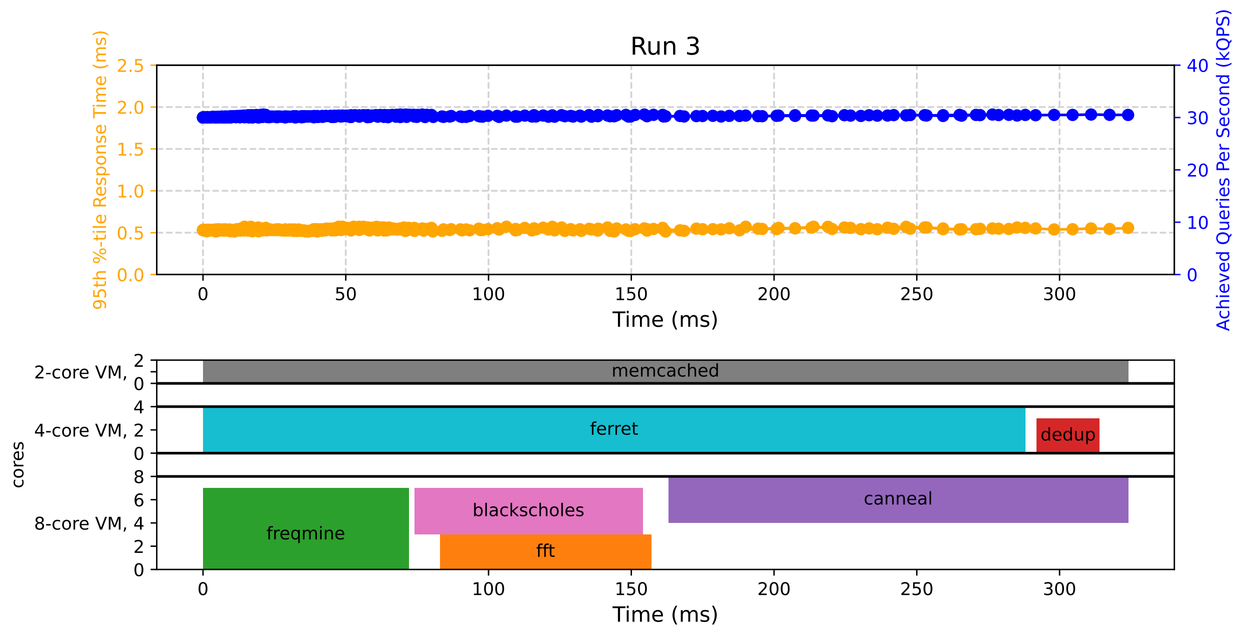


Figure 3: Run 3 of static scheduler

2. [**17 points**] Describe and justify the "optimal" scheduling policy you have designed. This is an open question, but you should at minimum answer the following questions:

   - Which node does memcached run on?
   - Which node does each of the 6 PARSEC apps run on?
   - Which jobs run concurrently / are collocated?
   - In which order did you run 6 PARSEC apps?
   - How many threads you used for each of the 6 PARSEC apps?

   Describe how you implemented your scheduling policy. Which files did you modify or add and in what way? Which Kubernetes features did you use? Please attach your modified/added YAML files, run scripts and report as a zip file. **Important: The search space of all the possible policies is exponential and you do not have enough credit 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.**

The scheduling policy that we considered ”optimal” was chosen based on different measures and factors that we collected in previous parts of the project.

First of all, we prioritized reaching the SLO level for memcached. Because of that, we decided to allocate it on the 2-core node. The measures we took without interference in Part 1 were obtained in a HighCPU-n2 machine with 2 cores, the same type of machine we encounter in this part. In our previous results we found that the SLO level can be reached with a margin of 0.75 ms, which we found appropriate for this part of the project. Moving memcached to a non-high-CPU machine may have proved risky as memcached was previously found to be sensitive to CPU interference implying it is a CPU heavy application. We assigned both cores with 2 threads to memcached.

As we did not have much information about the effect of collocating various PARSEC jobs with memcached, we favoured a conservative approach by isolating memcached on its own machine. The benchmark we previously identified as the best candidate for collocation with memcached was canneal, but canneal is a long task that benefits greatly from parallelization, and thus should not be limited to sharing the two core machine.

To schedule the different jobs we considered three main aspects from previous parts of the project: the absolute run time of each application, the scalability with number of threads and the resource utilization of each job. These measures (run time, number of threads and resource sensitivity) were obtained on a 8 core machine in Part 2. It is important to point out that the number of threads were chosen based on the relative relationship with the number of cores. The explanation will be done per each available machine:

We decided to run 2 jobs on the 4 core, high memory machine: ferret and dedup. Ferret has the highest memory utilization and it scales well with the number of threads (it reduced from 12 minutes to 3 minutes with up to 12 threads in our previous measurements). We decided to assign 4 cores and 4 threads and run it first. Dedup was the shortest one. Its high utilization of memory and its positive scalability with the number of threads were the key factors to run it alone after ferret.

The rest of the jobs were run in the 8 core standard machine. Freqmine scales very well and takes a lot of time to finish, so we decided to assign up to seven cores and six threads. For fft we had no information about the scalability with number of threads, so we chose 2 threads in 3 cores. It is a potential choice for running parallel applications. In the same situation we found blackscholes with medium sensitivity for most resources. It required more resources, so we assigned five cores and five threads given the positive response from threads scalability. That is why we decided to run blackscholes and fft at the same time. Finally, canneal scaled well up to three threads. Its cpu utilization is not very high, so we assigned in the end four cores (given that it was running alone in the end and the comparison with previous results was improving). We ran it alone so that other jobs did not interfere with it.

This policy was implemented following different steps. First of all, the yaml files for each job were edited. It was important to select the node to which the job was going to be deployed in the field *nodeSelector*. Also, inside the arguments we added the preferences for the specific cores *(-c)* and the number of threads (*-n*). When various jobs were deployed to the same node at the same time, we added the resources limits to avoid interfering among them (*resources, limits, cpu*). It is important to add that the yaml for memcached was also assigned to one specific node.

To run all the PARSEC applications our first approach was to create the jobs with the *kubectl* command and let them run automatically. We reached a mean time of 423 seconds and we found out that the applications were not running on the sequence we were expecting. We naively assumed that the jobs were going to be deployed in the suitable order we came up with. In other words, we misunderstood cpu affinity and we decided to deploy a static scheduler with conditions on when the previous job were to finish. The static scheduler can be found in the attached zip file (**static_scheduler.sh**). Following the strategy explained above, we deploy the new jobs when the previous ones were completed by checking the job status every second. By changing to the second policy the time decreased around 100 seconds and its variability was much lower.

For the 4 core node the approach was very straightforward: we directly ran ferret. When it had finished, dedup started. For the 8 core node there is more variability in terms of possibilities. Firstly, we decided to run freqmine and fft. However, its effect was negative to freqmine, so we postponed it until freqmine was completed. The conditions for this case were that freqmine finished before blackscholes started, and also balckscholes (and consequently freqmine) before canneal.

After observing the results, a potential improvement could be to increase the number of cores of canneal or freqmine reaching the limits of the whole core. Also, we noticed that the starting of each job takes some seconds. Although likely difficult to implement, starting a job slightly before its predecessor completes could remove this delay. Another way to decrease this delay would be to alter the delay parameter determining the time between executions of the scheduler's. This way the scheduler would notice a job had completed faster netting us a slight performance boost.

Part 4 [76 points]

1. **[18 points]**

   For this question, use the following `mcperf` command to vary QPS from 5K to 120K:

```
$ ./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 5000:120000:5000
```

   a) 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.

   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.

The requested plot for this subquestion is included in Figure 4. To summarize the results, the only set of parameters at which memcached both achieved the target QPS and respected the SLA were $T = 2$, $C = 2$. In contrast, in the case where $T = 2$, $C = 1$ neither of these goals were achieved. The cases where $T = 1$, $C = 1$ and $T = 1$, $C = 2$ exhibit middling performance. Both fail to satisfy demanding requests with the two core case performing slightly better. Both also achieve comparable latencies at the top of their respective achieved QPS ranges.

From the results of our plot, we can conclude that the number of cores has a greater effect on the performance of memcached, as both two core cases outperform both single core cases. To be able to satisfy requests on the higher end of the range, both two threads and two cores are necessary. Furthermore, when threads outnumber cores we see a sharp decrease in performance. This intuitively makes sense, as the threads are competing with each other for the same set of resources and there is an increased overhead compared to the single thread single-core case. This is the only case where the SLA is breached in addition to memcached not being able to reach the desired QPS. The performance of the $C = 1$, $T = 1$ and $C = 2$, $T = 1$ cases are similar, implying that to fully leverage the effect of added cores, multiple threads are needed. This is also logical as a single thread can only be running on one core at a given time.

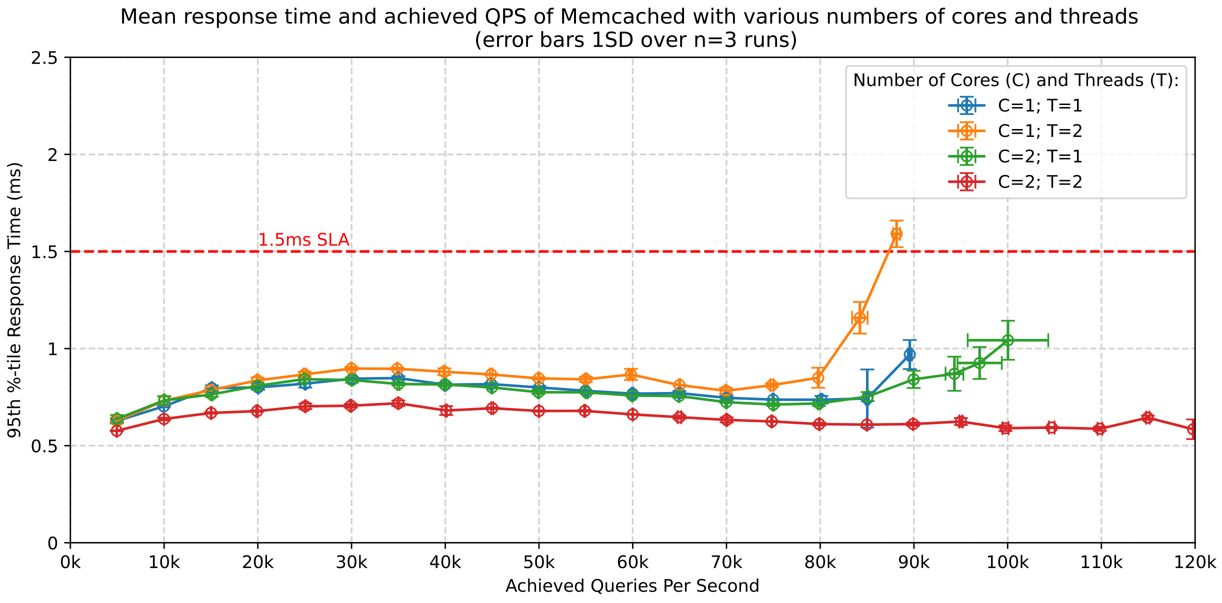


Figure 4: Memcached performance with various numbers of cores and threads. Target QPS ranges from 5k to 120k in intervals of 5k.

b) To support the highest load in the trace (120K QPS) without violating the 1.5ms latency SLO, how many memcached threads (T) and CPU cores (C) will you need? i.e., what value of T and C would you select?

To support the highest load in the trace we would select the parameters $T, C = 2$. The data shows that with parameters $T = 2, C = 1$ the SLA is likely to be violated. The reaming two sets of parameters tested fail to support a full load of 120k QPS, and are therefore inappropriate.

When $T, C = 2$, memcached supported the maximal target load without issue and without violating the SLA. In particular, we see that the achieved latency remains fairly constant at all loads tested. A higher allocation of CPU cores and/or threads is therefore unnecessary.

c) Assume you can change the number of cores allocated to memcached dynamically as the QPS varies from 5K to 120K, 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 1.5ms 95th percentile latency SLO while the load varies between 5K to 120K QPS? i.e., what values of T would you select?

We propose setting the number of threads to $T = 2$. As previously discussed, to support the higher end of possible workloads, both T and C must equal 2. However, this approach has a disadvantage, namely, if we wished to use more cores for other jobs, memcached would end up running in the $T = 2, C = 1$ case where its performance is worst and it can only support relatively light loads. The potential harms of this issue can however be mitigated by a smart scheduling policy.

d) Run memcached with the number of threads T that you proposed in c) above and measure performance with $C = 1$ and $C = 2$. 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 5K to 120K. In each graph, use two y-axes. Plot the 95th percentile latency on the left y-axis. Draw a dotted horizontal line at the 1.5ms 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.

The requested plots are included in Figures 5, 6 on the following page.

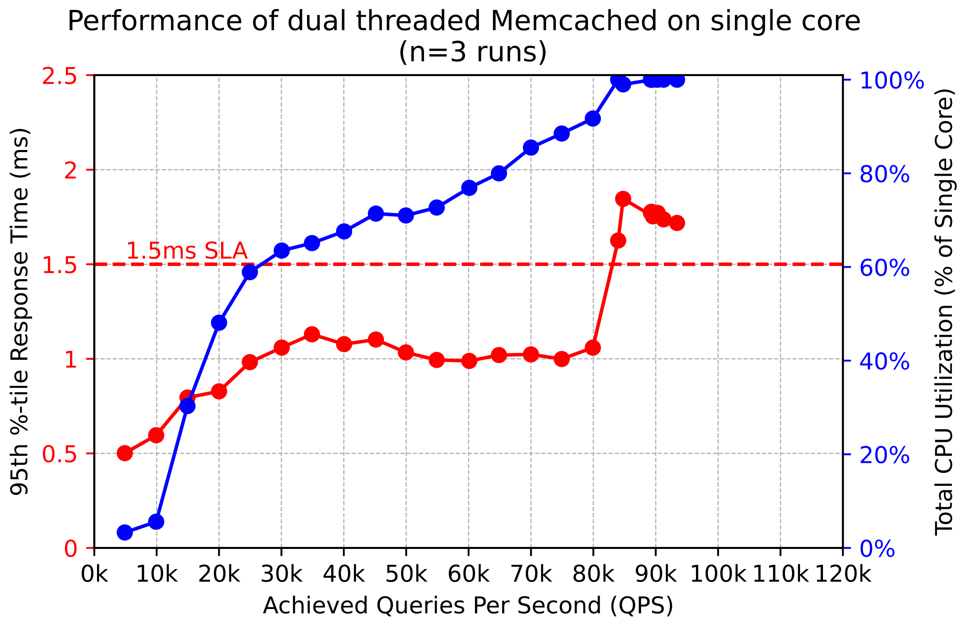


Figure 5: Single core CPU utilization and 95th percentile latency of memcached. Target QPS ranges from 5k to 120k in intervals of 5k.

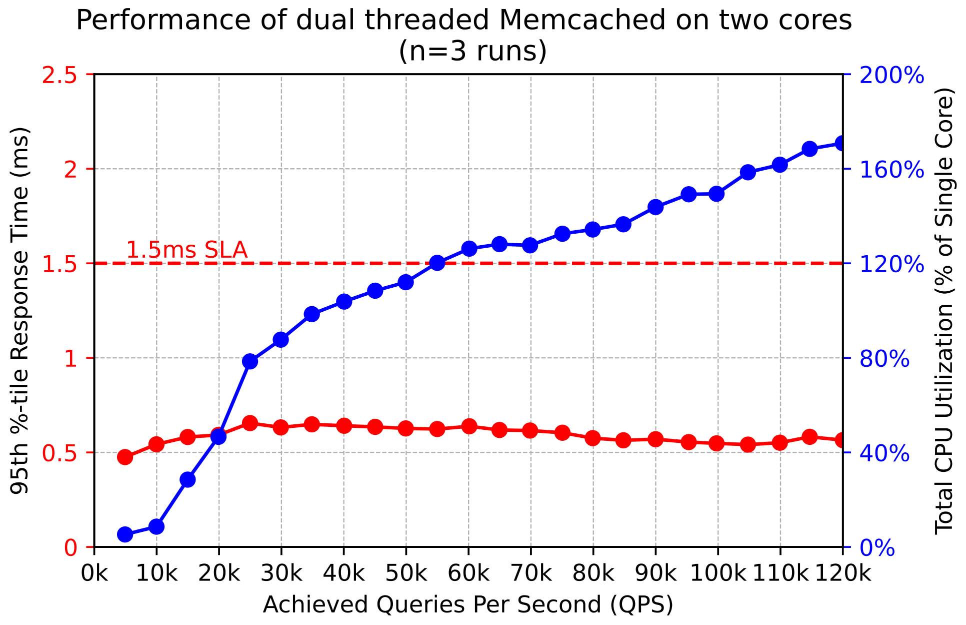


Figure 6: Dual core CPU utilization and 95th percentile latency of memcached. Target QPS ranges from 5k to 120k in intervals of 5k.

2. [**15 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.

   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 1.5ms 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.** 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. To describe your scheduling policy, you should at minimum answer the following questions. For each, also **explain why**:

   - How do you decide how many cores to dynamically assign to memcached?
   - How do you decide how many cores to assign each PARSEC job?
   - How many threads do you use for each of the PARSEC apps?
   - Which jobs run concurrently / are collocated and on which cores?
   - In which order did you run the PARSEC apps?
   - How does your policy differ from the policy in Part 3?
   - How did you implement your policy? e.g., docker cpu-set updates, taskset updates for memcached, pausing/unpausing containers, etc.

Scheduler Design

The four core modules/components of our dynamic scheduler are presented in Figure 7. These are the logical division of the CPU into *core blocks*, the *memcached finite state machine (FSM)* responsible for controlling the execution of memcached, the three *job queues* responsible for categorizing, (partially) ordering and prioritizing parsec jobs and various data structures for storing *static parameters* and the *execution log*. We will explain our scheduler's architecture and justify the key design decisions taken by exploring each of these components in turn. For easier reading/assessment, a summary of the key design decisions requested in this section's instructions can be found in Appendix A.

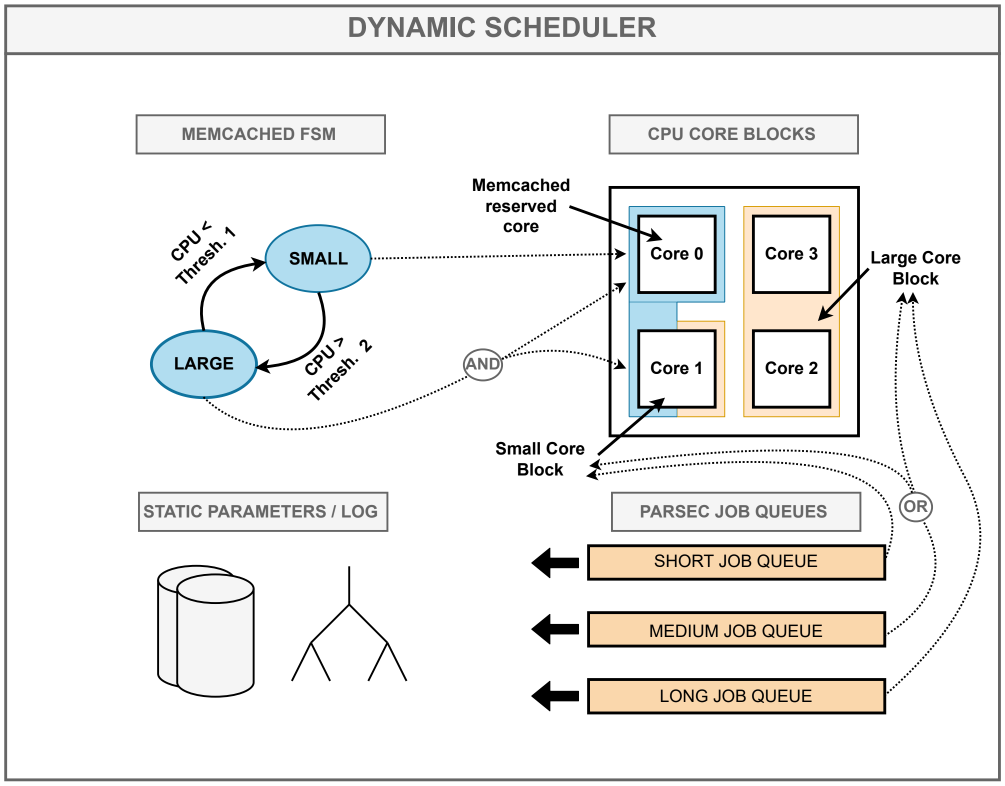


Figure 7: Logical Model of Dynamic Scheduler

Core Blocks

The decision was made to split the CPU into three separate logical units; the large core block, small core block and memcached reserved core. Each block is a separate unit of compute, and with the exception of memcached (which can run on the reserved core and small core block simultaneously), jobs cannot be run between core blocks and are never collocated on them. This arrangement provides an acceptable degree of flexibility when deciding how much processing power to assign jobs, while guaranteeing isolation. Specifically, this helps eliminate interference between parsec jobs in resource categories that are not shared between cores (all except last level cache and memory bandwidth), and avoids the dangers of collocating jobs on cores running memcached.

A visual reference of the different core blocks can be seen in Figure 7. The following is a summary of each core block's size, role and behaviour:

- **Memcached Reserved Core:** Core 0 of the CPU constitutes the memcached reserved core. As the name implies it is only ever used to execute memcached. Depending on the state of the memcached FSM, it will either run memcached alone or together with the small core block.
- **Small Core Block:** The mall core block corresponds to CPU core 1 on the host machine. It is used to execute both parsec jobs and memcached in a serial fashion. When it is needed to handle a heavy memcached workload, it will pause the parsec job it is currently running. This job will be resumed when memcached releases the small core block's resources. As it only consitutes one core and is not available for pasec all the time, the small core block will preferentially fetch jobs from the short job, as these are shorter and do not scale as well as

others. If no short job is available, it will attempt to fetch jobs from the medium and long job queue respectively. When no more jobs are available (and a job is running in the large core block) the memcached FSM will bind the small core block to memcached for the remaider of the schedulers execution.

- **Large Core Block:** The large core block consists of CPU cores 2 and 3. It is meant for running long parsec jobs that benefit from parralelization and scale well with the number of cores/threads they are run on. Throughout execution this block runs parsec jobs sequentially and fetches jobs in order of preference from the long, medium and short jobs queues (explained in more detail in following sections). As the larges unit of compute, this block is utilized as much as possible and will fetch jobs from the small core block rather than remain idle.

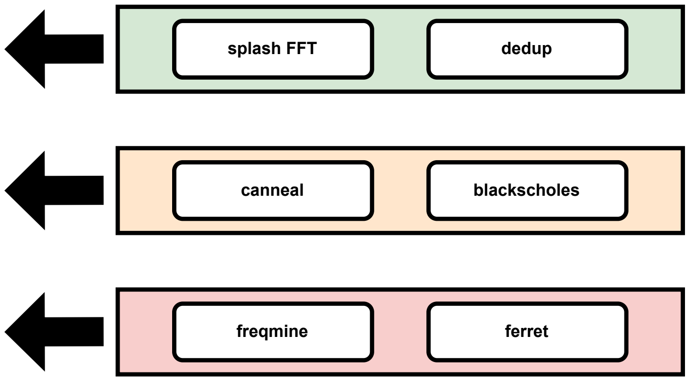
The motivation for this design is as follows. For reasons discussed in the previous sub-question, we decided memcached should be ran with two threads of execution. Given this parameter, the data shows that two cores are necessary to serve high workloads without violating the SLA. To improve overall utilization (and to stop our scheduler being very boring) we chose to switch the additional core needed for memcached between jobs. Our architecture provides the necessary degree of flexibility by having a single core block that can be assigned to memcached when needed.

We also observed that most parsec jobs would likely compromise the SLA if collocated with memcached. In addition to this the interference profiles of the parsec jobs added an additional layer of complexity to consider when determining execution order and introduced a risk of exponentially increasing the scheduler run-time if a certain resource were to run out. To a large extent, all of these issues are solved by our decision to prioritize isolation. This comes as a tradeoff, as this likely leads to some loss of efficiency due to idle CPU while a job is busy with I/O. However, this appears to have payed off as our schedulers performance is well within outlined parameters (presented in the following sections).

In terms of the effect of these decisions on efficiency, since no block is left idle while jobs are queued, our architecture still achieves high utilization. The theoretical maximum waste is equal to the compute power of a single core for the duration of a single job (as we shall see this jobs will be medium length at most) and only for the proportion of time that memcached doesn't require two cores to run.

Job Queues

The parsec jobs were divided into three job queues based on their length and scalability. These determine what core block is preferred for each job (consequently how many resources it gets) and what order they are likely to run in. This way, longer jobs that scale better can take better advantage of the large core blocks multiple cores, and can consequently be run with a higher number of threads. The opposite logic applies for jobs in the short job queue. The medium queue acts as a middle ground, and it's jobs are assigned to whatever core block runs out of preferred jobs first. A detailed description of each queue/role as well as the specific allocation of jobs to queues used can be found in Figure 8.



CATEGORY	DESCRIPTION	CORE ALLOCATION PREFERENCES
SHORT JOB QUEUE	Short jobs that scale badly or who's scaling properties are not known	Prefer to be executed on **small** core block
MEDIUM JOB QUEUE	Middling jobs that do not fit clearly into either category	Are executed after jobs from short and long queue on **whatever block has space**
LONG JOB QUEUE	Long jobs that scale well	Prefer to be executed on **large** core block

Figure 8: Description of Job Queues

Note that allocation to a certain queue represents a preference for a certain core block rather than a strict requirement. For example, a task remaining in the short queue when all medium and long tasks are exhausted may be assigned to the large core block and vice versa. Tasks placed at the front of the short and long queues are certain to start execution on their preferred block. With this in mind, we determined that a stronger preference is better for freqmine and splash than dedup and ferret which motivated the ordering presented in the figure above.

Since in our architecture, tasks are taken from the long and short queues first, the last task in the medium queue will be the last task to execute overall. If at any point in its execution it remains the only job left running, it will be moved to the large core block so as to speed up execution. It is, therefore, more likely to spend more time in the large core block than its predecessor. Based on this we decided blackscholes should be placed at the back of the medium queue and canneal at the front.

Memcached FSM

The final active component of our scheduler is the memcached FSM. It is responsible for monitoring the performance of memcached and switching it between two states. The SMALL state is selected when memcached is given a smaller workload that it can handle with a single CPU core. The LARGE state is entered when memcached requires the additional core of the small core block to not violate the target SLA. In this case the parsec job currently in the small core block is suspended untill the cpu core is released by the FSM. The thresholds for changing between these two states are a tunable parameter of the scheduler. Decreasing the threshold for moving from the LARGE to the SMALL state improves overall execution time at the risk of a higher number of violations. Conversely, increasing the threshold from SMALL to LARGE lowers the risk of SLA being violated but increases the overall run time.

An important consideration is to keep some distance between these two thresholds to prevent thrashing, although in practice due to the nature of the changes in CPU load we were unable to fully avoid this issue. By testing a limited range of possible thresholds, we determined that a single core utilization over 60% should move memcached form the SMALL to the LARGE state, and dual core utilization under 130% should move it back to the SMALL state.

Static Data and Log

The final component of our scheduler is the static data associated with a run. This includes the thread allocations for each parsec job presented in Table 1. Two main factors were considered when determining the optimal number of threads. The first was what queue each job was allocated to and consequently what number of cores it would likely run on. The second was the experimental observations of how each job scales with the number of threads. For example, since both short queue jobs will likely run on one core, and since the scaling of one is not known and the other exhibits anti-scaling both were assigned a single thread of execution. The numbers selected are somewhat conservative to avoid the risk of decreased performance when threads outnumber cores.

Parsec Job	No. Threads
dedup	1
splash FFT	1
blackscholes	2
canneal	1
freqmine	4
ferret	4

Table 1: Thread allocations of parsec jobs

In addition to the thread allocations, several tunable parameters are stored by the scheduler. These include things like the starting state for the memcached FSM, the interval between executions of the scheduler's main loop and FSM thresholds. The scheduler also maintains a log of parsec end states, start and end times and timestamps of when these were paused. An execution trace of the scheduler's behaviour useful for debugging is outputted to users, but not saved.

Scheduler Implementation

The main body of our scheduler consists of around 300 lines of python code. In addition to this, several bash scripts were used to aid with running the experiments. In particular, we found it beneficial to limit the execution of the scheduler to the cores of the large CPU block with the *taskset* command. When collocated with memcached it caused enough interference to noticeably increase the violation rate.

Several wrapper functions were written to interface with the parsec jobs and control their and memcached's CPU affinity (included in the *utility.py* file). Jobs were launched, paused, unpaused and retired in docker containers using the docker library for python. These were modified to continue existing upon completion so that their logs could be inspected and retrieved. The CPU affinity of memcached and the parsec jobs was changed using *taskset*. It should be noted that this required superuser permissions to be granted to the scheduler.

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 42
```

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. Also 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 datapoints with 95th percentile latency > 1.5ms, as a fraction of the total number of datapoints.

job name	mean time [s]	std [s]
dedup	166.3	27.9
blackscholes	403.0	99.8
ferret	962.3	48.1
freqmine	481.7	24.9
canneal	728.3	66.4
fft	208.3	5.4
total time	1569.8	25.4

Run	SLO violation ratio for 10-seconds interval (%)
Run 1	3.33 %
Run 2	2.66 %
Run 3	1.67 %

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.

The requested plots are included below:

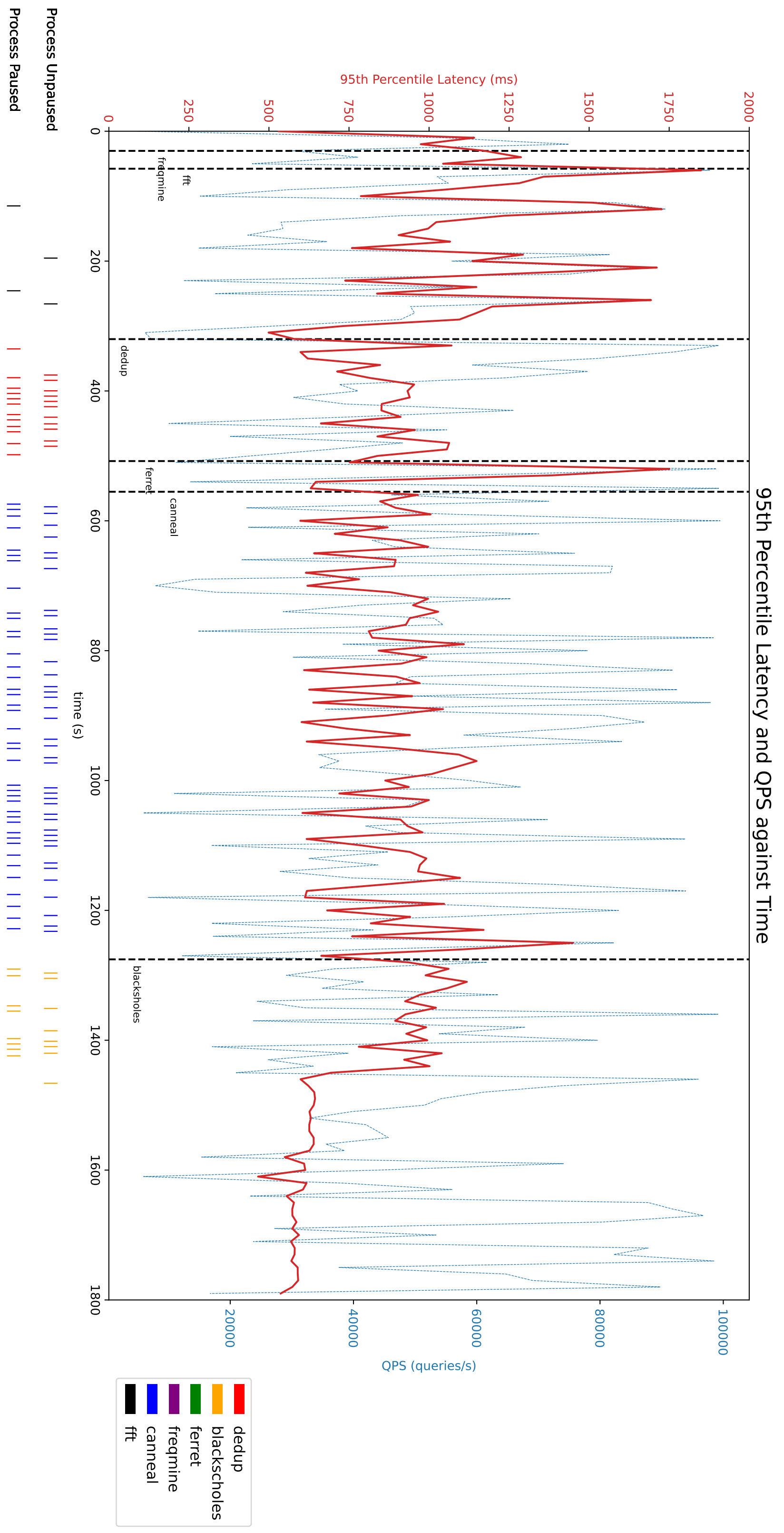


Figure 9: Plot 1A

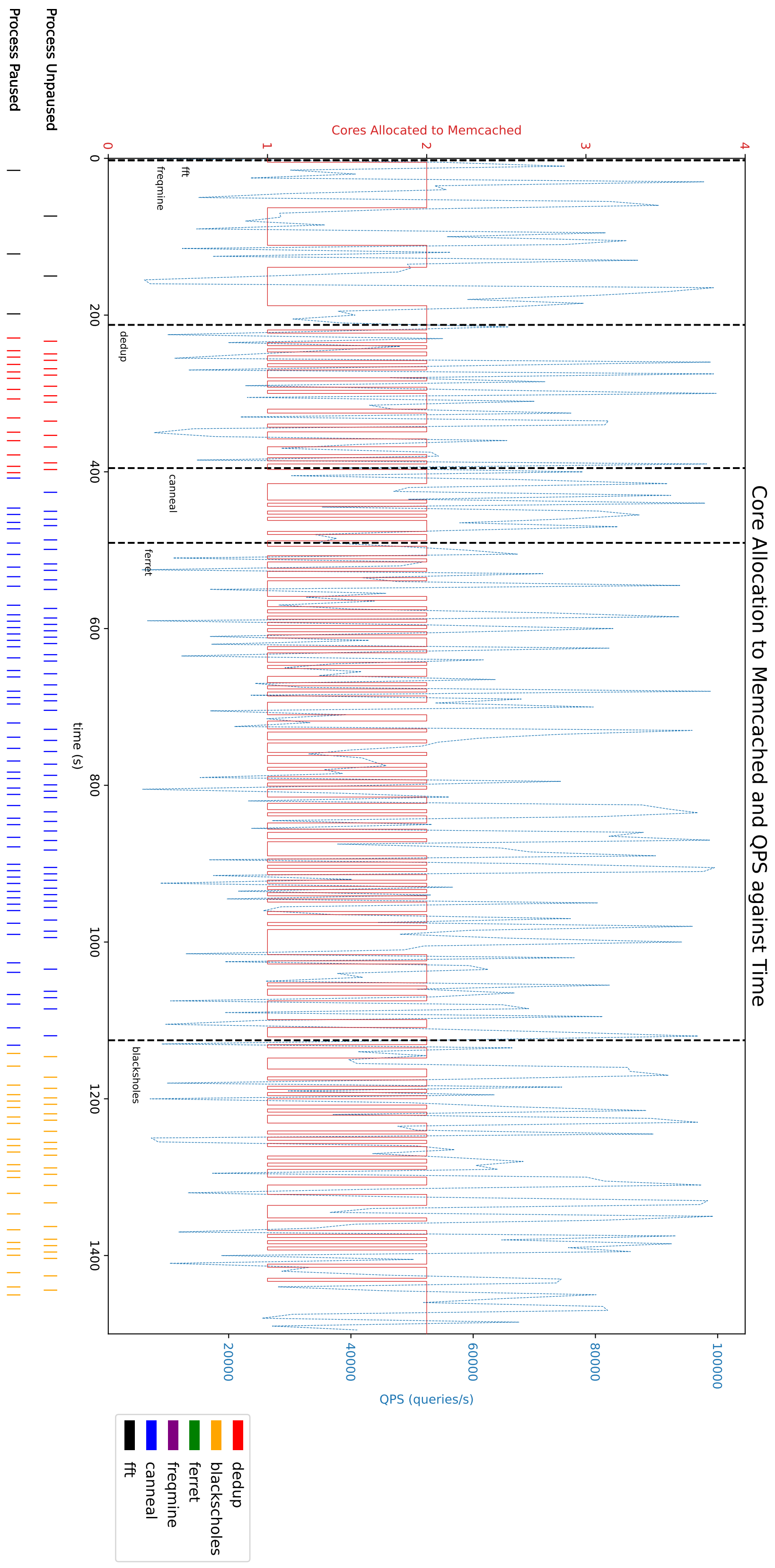


Figure 10: Plot 1B

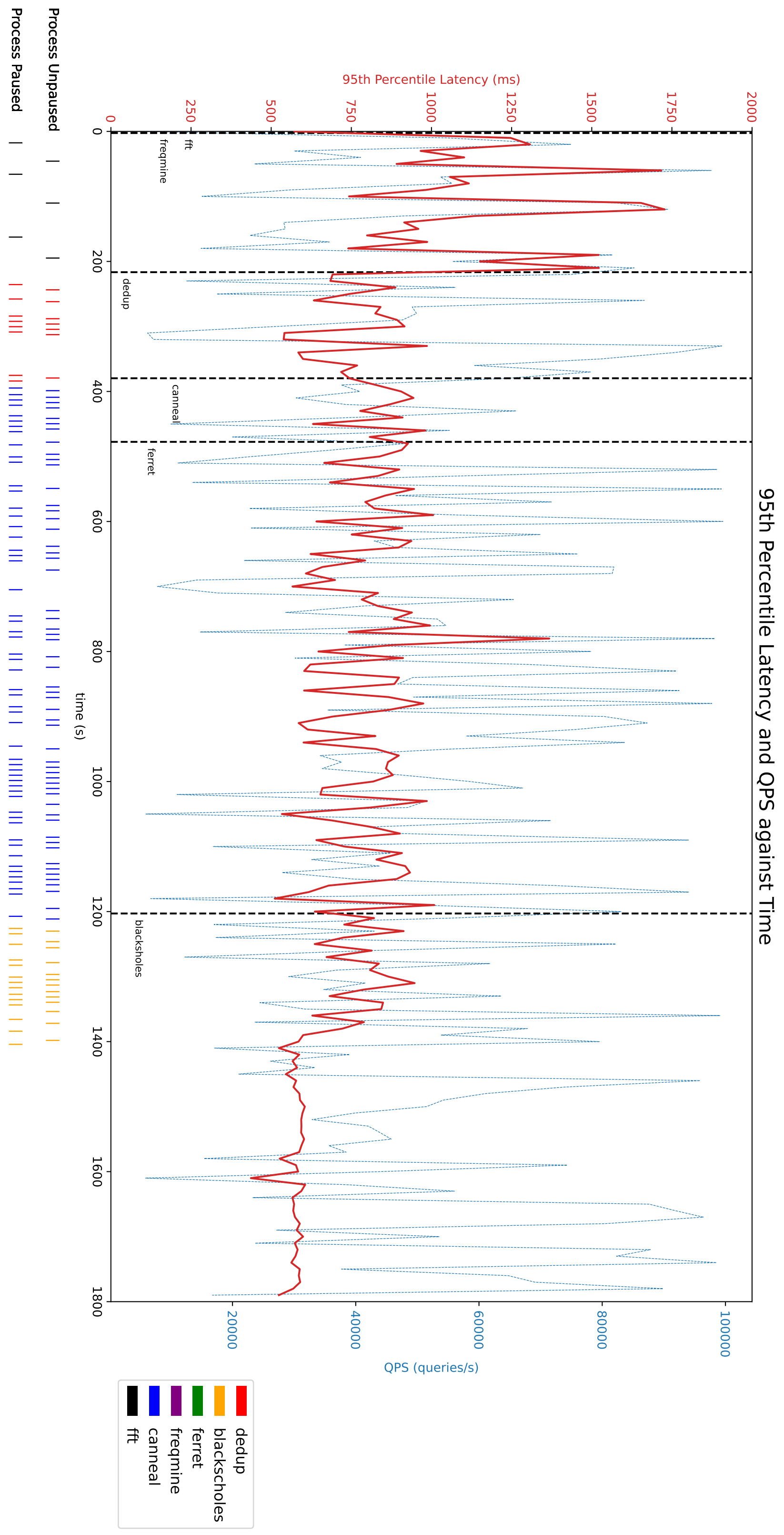


Figure 11: Plot 2A

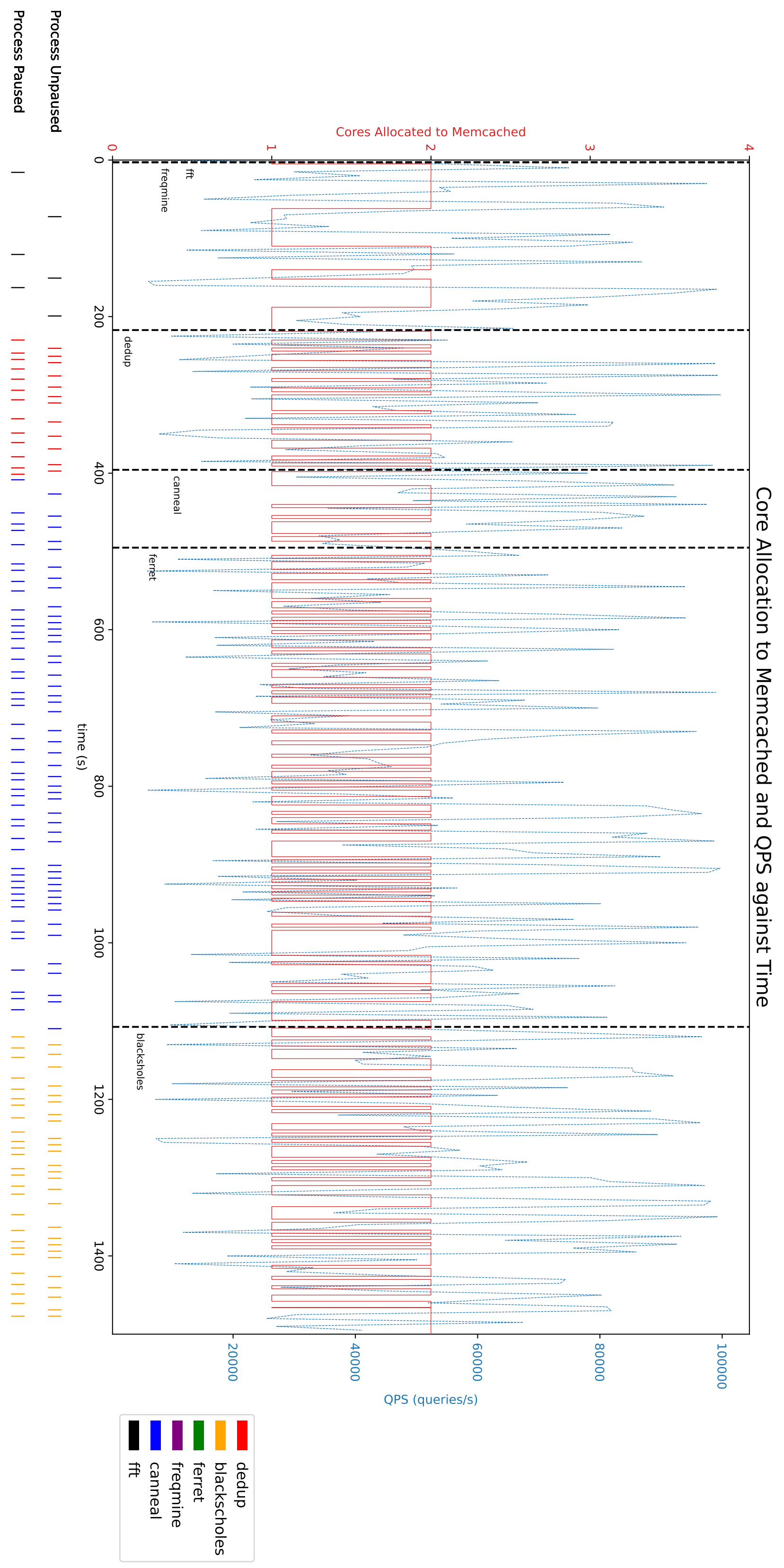


Figure 12: Plot 2B

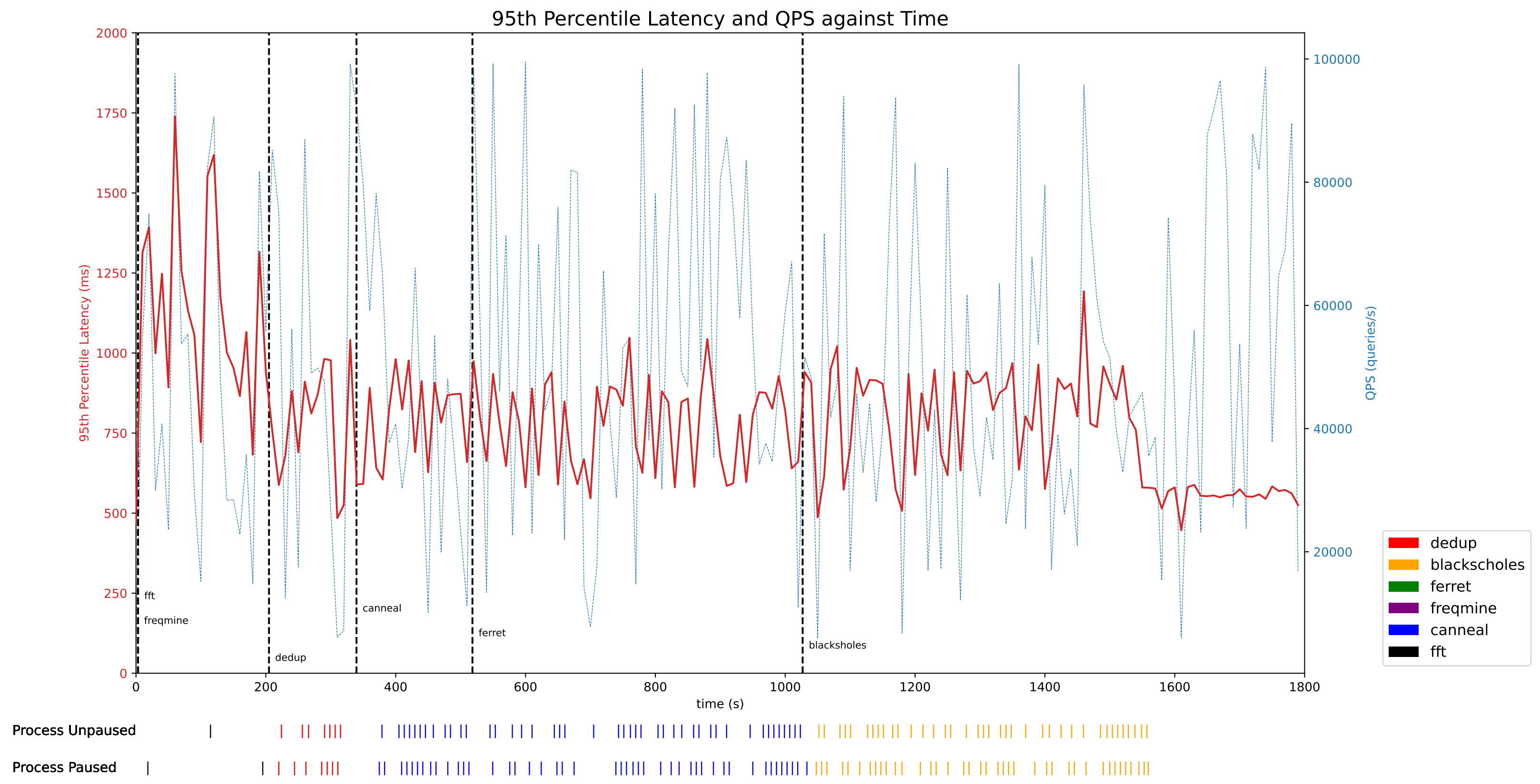


Figure 13: Plot 3A

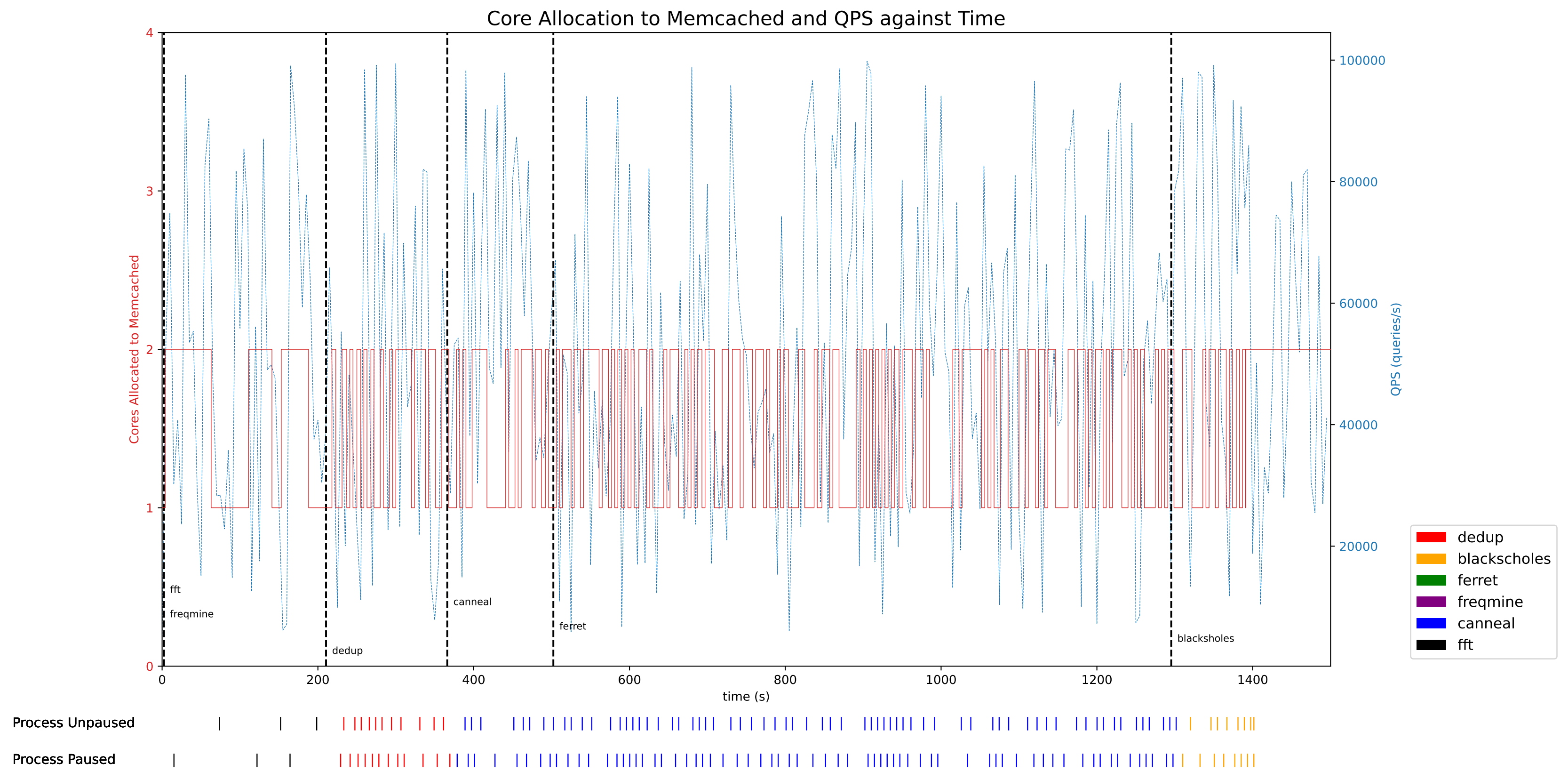


Figure 14: Plot 3B

4. **[20 points]** Repeat Part 4 Question 4 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 42
```

You do not need to include the plots or table from Question 4 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 4. What is the SLO violation ratio for memcached (i.e., the number of datapoints with 95th percentile latency > 1.5ms, as a fraction of the total number of datapoints) 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%? 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]
dedup	171.7	11.9
blackscholes	397.3	76.5
ferret	936.3	34.7
freqmine	493.3	5.0
canneal	790.0	98.5
fft	210.0	2.2
total time	1571.3	9.0

Run	SLO violation ratio for 6-seconds interval (%)
Run 1	2.67 %
Run 2	3.00 %
Run 3	2.67 %

The SLO for the 5-seconds interval is 3.05%. We can observe that even though the load variability has increased, the violating rate has remained low. This shows that our scheduler is more or less stable when varying the QPS interval.

Based on the fact that the SLO violation for 10-seconds interval was 2.55% on average and the one for 5-seconds interval was 3.05% we decided on a 6 second interval. Based on the observed stability of the SLO we were reasonably certain this would achieve the desired SLO level of 3%. After 3 runs, the SLO violation was on average 2.78% with a standard deviation of 0.15%. The requested plots for the 6 second interval are included in Figures 15-20 on the following pages. To conclude the project and the explanation of the dynamic scheduler, we would like to add some comments of how we could improve the performance of the scheduler.

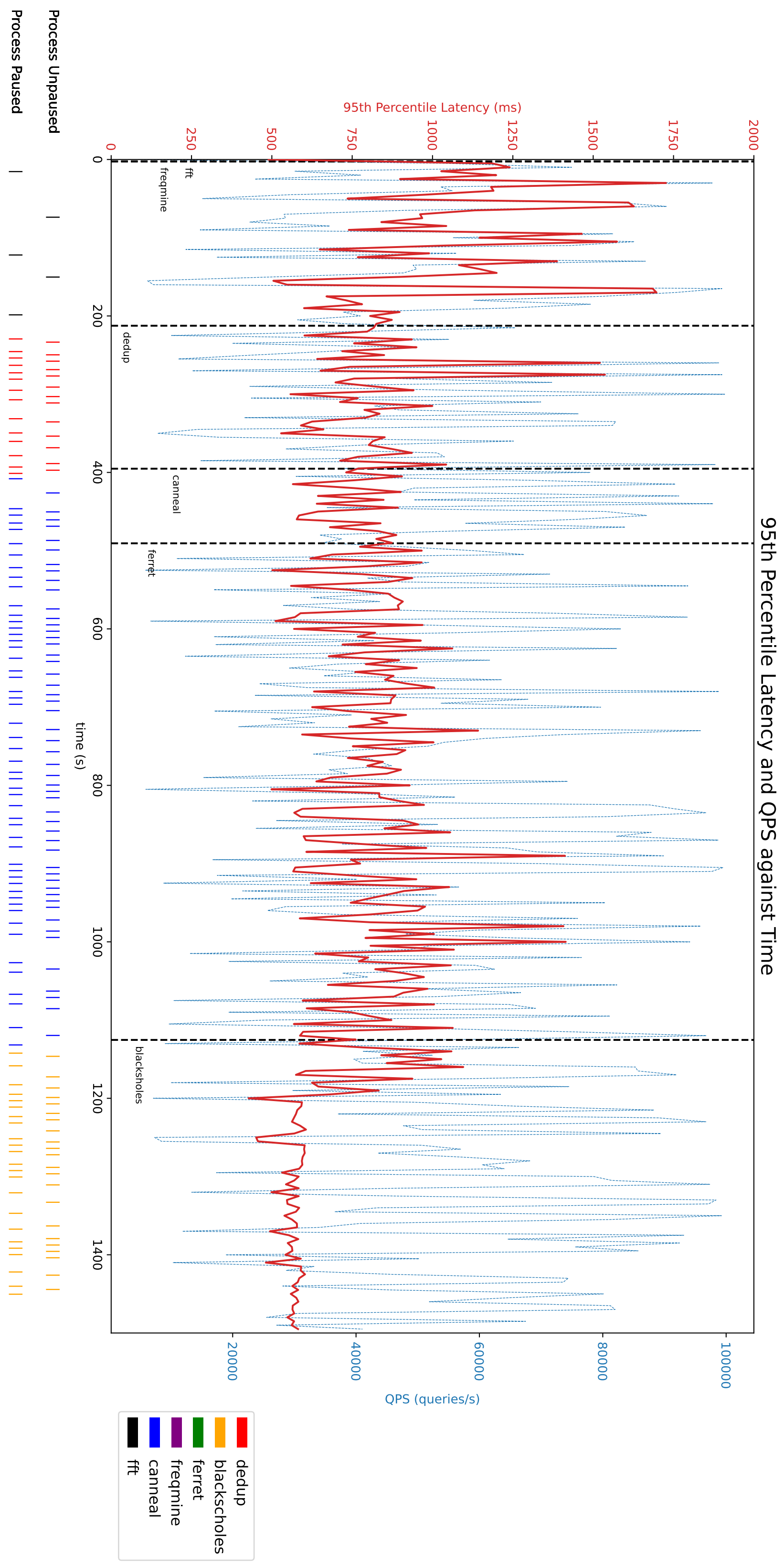


Figure 15: Plot 1A

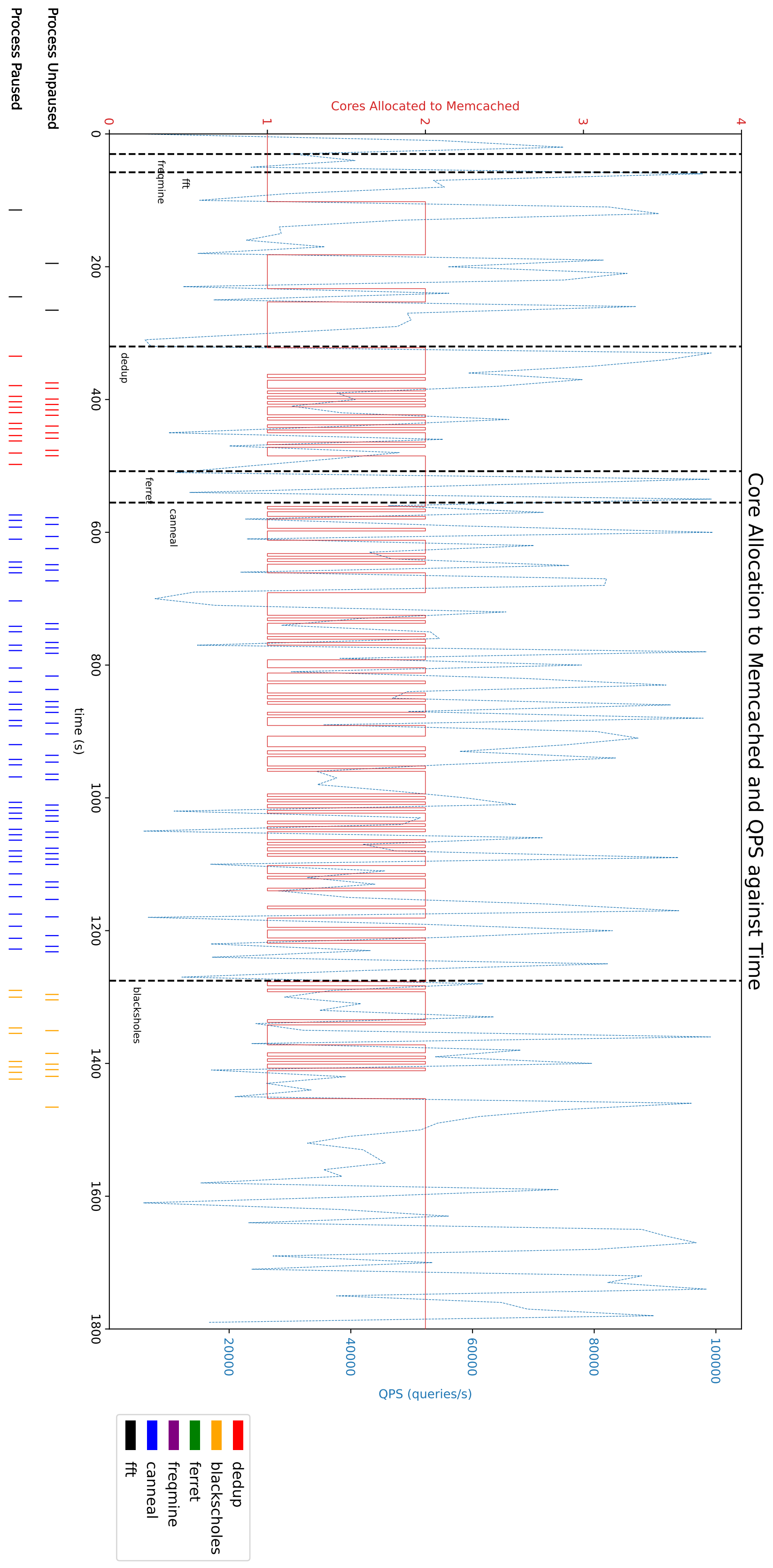


Figure 16: Plot 1B

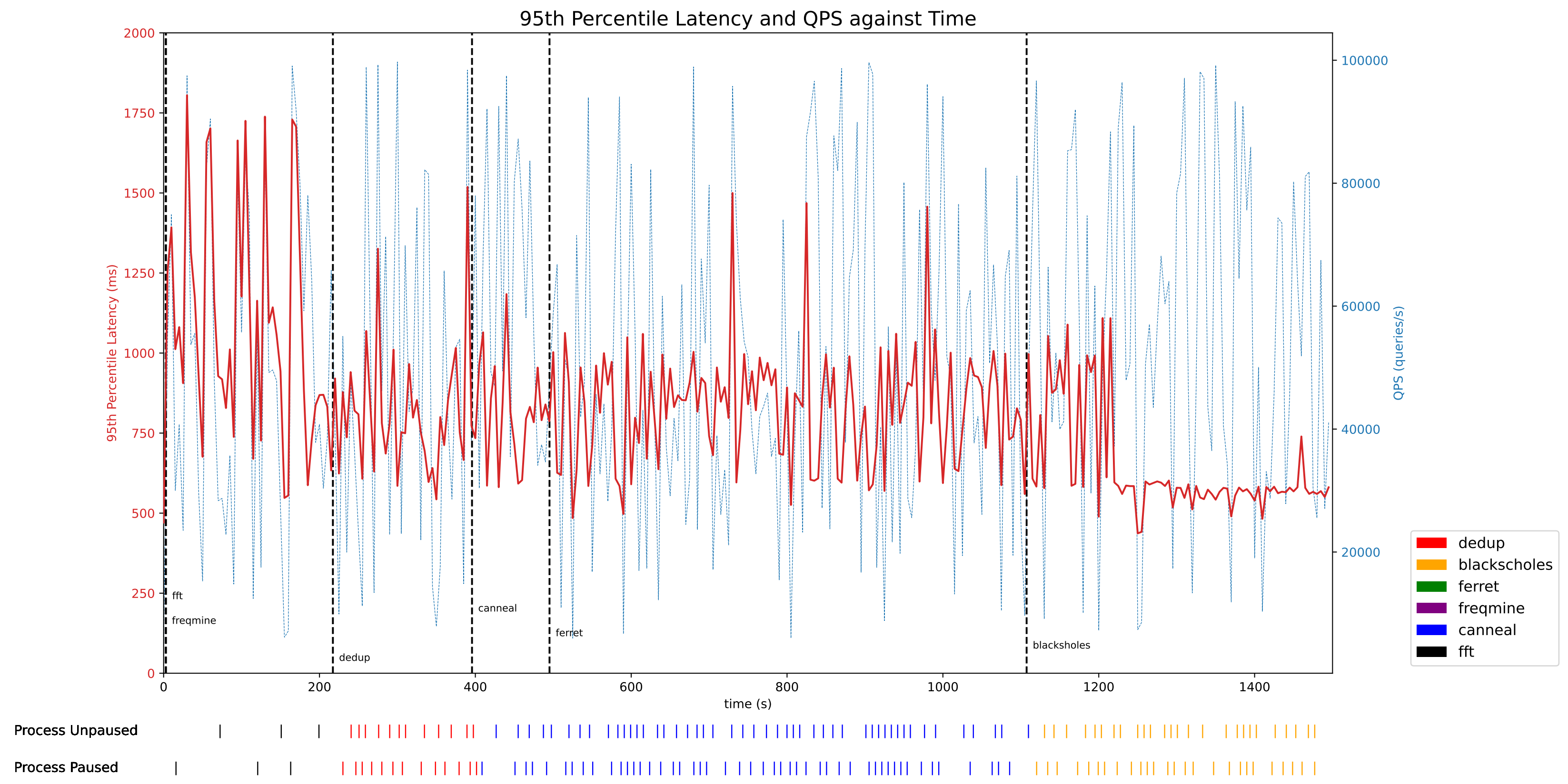


Figure 17: Plot 2A

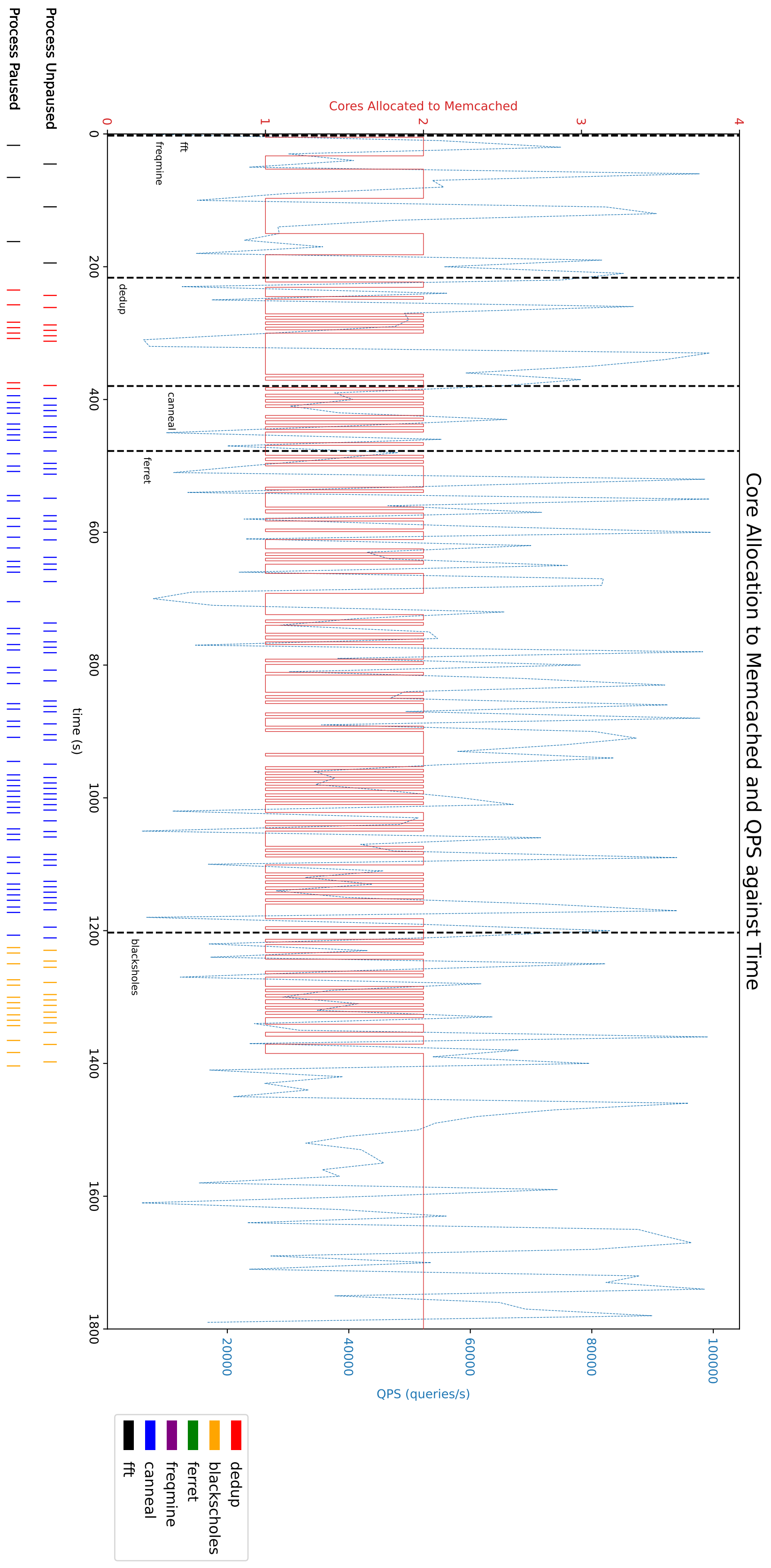


Figure 18: Plot 2B

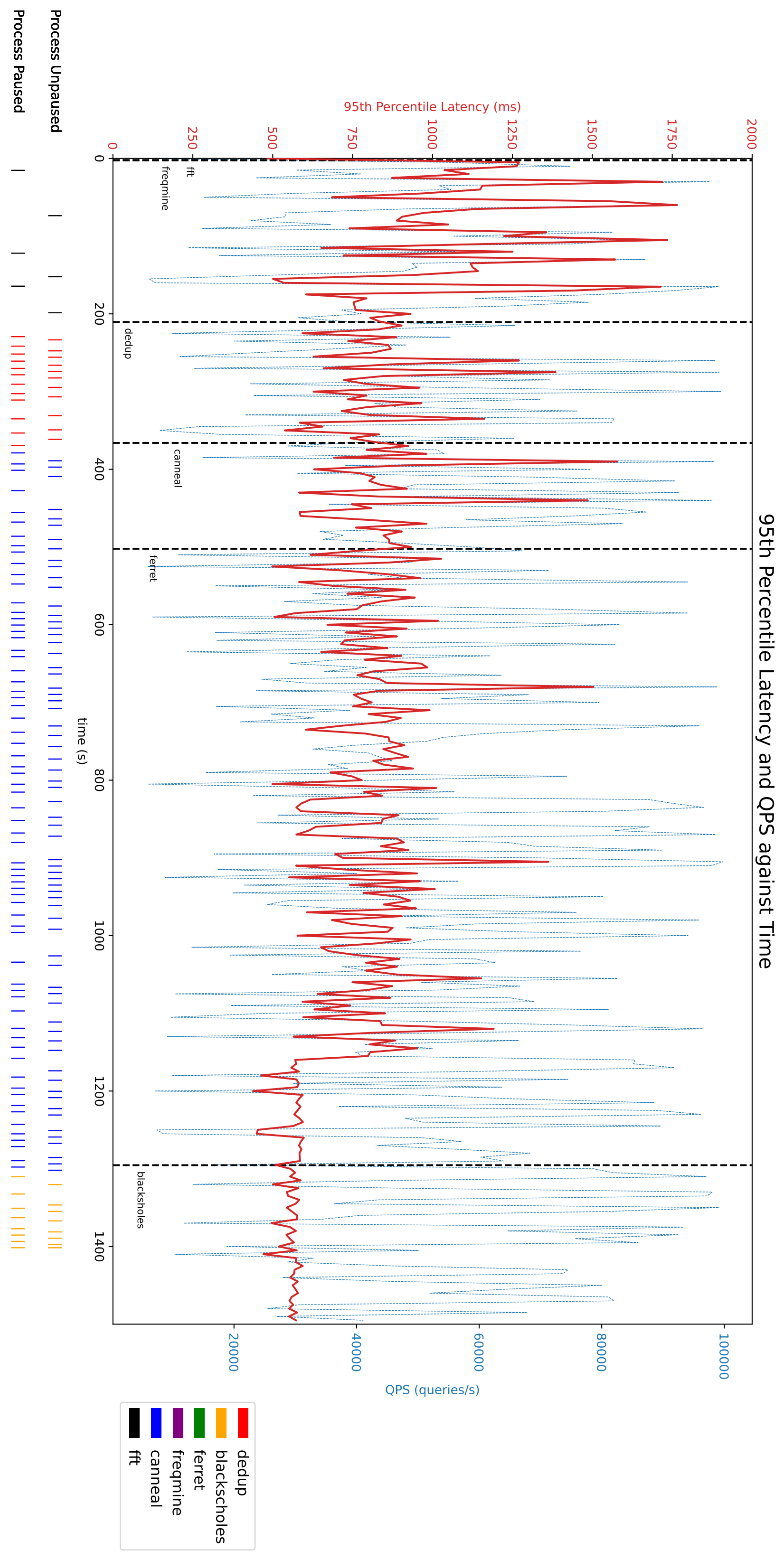


Figure 19: Plot 3A

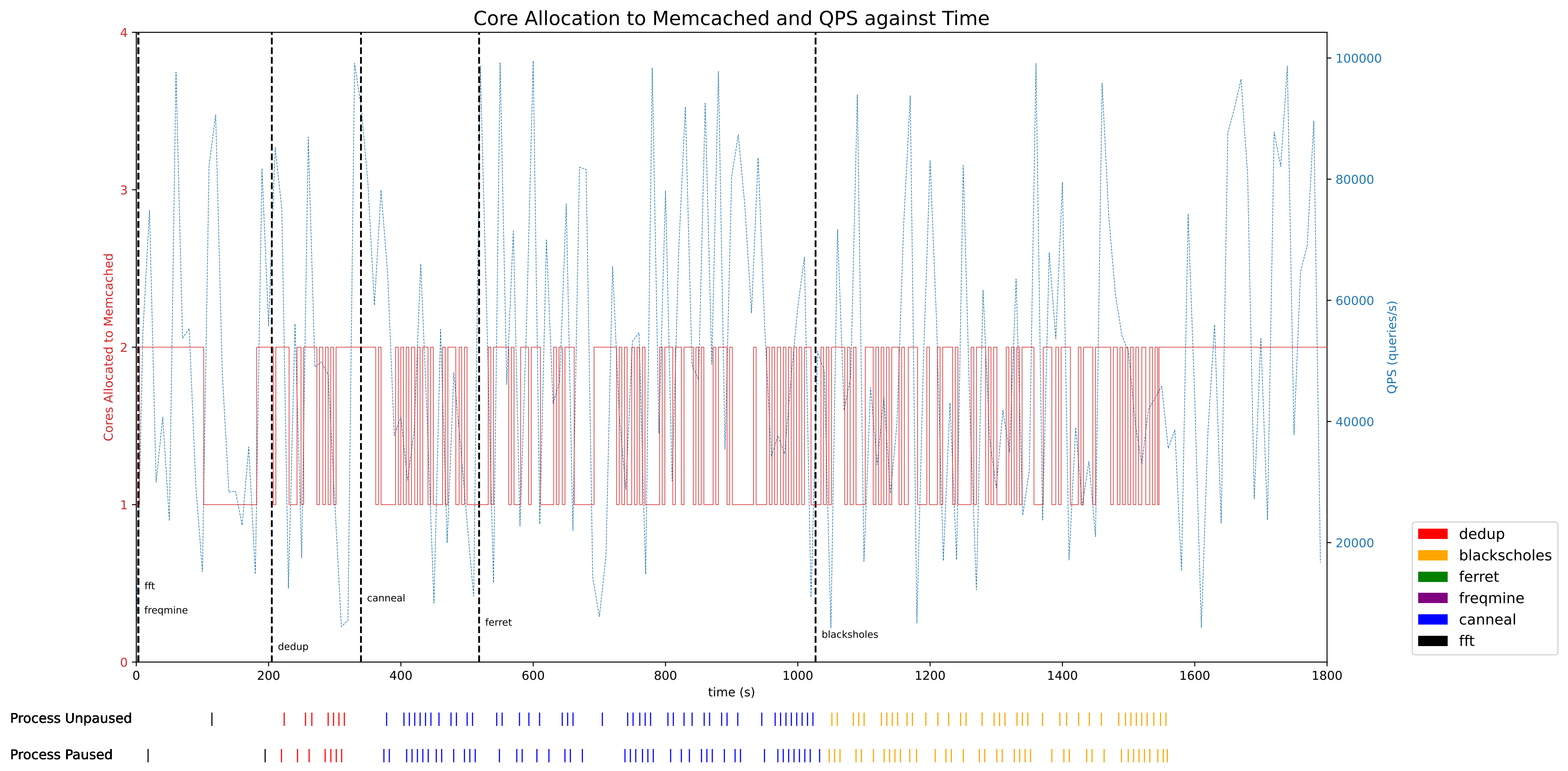


Figure 20: Plot 3B

Appendix A: Design Decisions for Dynamic Scheduler

The following table contains a summary of the major design decisions that went into our team's dynamic scheduler. It is intended to serve as a summary of our answers to the instruction questions for Part 4.2. All of these can be found explained in more detail in the main text of this report.

Question	Design Decision	Motivation
How do you decide how many cores to dynamically assign to Memcached?	Between 1 and 2 cores.	As discussed in Section 4.1, 2 Cores and 2 Threads appear to be the minimum needed to serve certain loads. Using 2 cores the entire time would be a waste of resources, so some of the time one is requisitioned for running parsec jobs.
How do you decide how many cores to assign each PARSEC job?	Parsec jobs are assigned 1 or 2. Specific allocations differ between runs.	Jobs that appeared to scale well with the number of threads and that took longer to execute (based on results from Parts 1 and 2) were preferentially assigned 2 cores. The opposite is true of jobs that scaled badly or took a very short time to execute. No jobs being assigned 3 cores is a limitation of our architecture and the decision to maximize isolation between various parsec jobs.
How many threads do you use for each of the PARSEC apps?	Differs per job. A summary can be found in Section 4.2 .	The number of threads was selected based on the observations of each job scaling property from Part 2 of the assignment. We also took into account how many cores a job would preferentially be run on.
Which jobs run concurrently / are collocated and on which cores?	All parsec jobs run on cores 2,3 and sometimes on core 1. Jobs are not collocated, instead, some of them are executed serially with Memcached. This allows us to take advantage of isolation between cores to eliminate the effects of most types of interference. What jobs run concurrently is not fixed between runs.	Jobs are not collocated to increase isolation and limit the effect of resource interference for resources that are core-specific.

In which order did you run the PARSEC apps?	Order not fixed. In an ideal scenario, splash would be run at the same time as freqmine, dedup with ferret and canneal with blackscholes. Regardless of changes in order, only two jobs will ever be executed simultaneously.	To maximize the schedulers utilization, the order in which jobs were run was not fixed. This way jobs that finish early or late due to random factors do not need to wait for/delay others. The order in which they run and the resources they are allocated are not completely random. Instead, our architecture induces a partial ordering and set of resource preferences for jobs. This way we can still maintain enough control to be reasonably certain that long/parallel jobs will run on more resources, and that we likely aren't left with a single long job to execute. This latter case would lead to lower utilization of our chosen architecture.
How does your policy differ from the policy in Part 3?	In part three the order in which jobs are executed is fixed. Jobs also have more cores to execute on, and the number of these cores is fixed. In Part 4 both the number of cores available to parsec jobs or Memcached and the order in which parsec jobs run are flexible and decided based on observed conditions. during runtime.	This was done to allow Memcached to use as many cores as needed without wasting CPU time while under a low load. As a consequence, the number of parsec jobs ran at any one time also had to be dynamic. This made it impossible to fully predict a job's run time. By using multiple queues, we can prioritize specific types of jobs based on what resources become available. A side effect of this is that the ordering of jobs is not necessarily fixed as the queues are empty at different and unpredictable rates.
How did you implement your policy? e.g., docker CPU-set updates, taskset updates for Memcached, pausing/unpausing containers, etc.	Python, the docker library for python and the UNIX taskset utility.	We wrote a library of high-level functions that interface with docker through the python API and isolate Memcached and parsec to specific cores