Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Systems@ETH Zürich

Cloud Computing Architecture

Semester project report

Group 048
Luca Tagliavini - 23-955-461
Christopher Zanoli - 23-942-394
Rudy Peterson - 23-941-701

Systems Group
Department of Computer Science
ETH Zurich
March 26, 2024

Instructions

- **Please do not modify the template!** Except for adding your solutions in the labeled places, and inputting your group number, names and legi-NR on the title page.

  Divergence from the template can lead to subtraction of points.

- Parts 1 and 2 of the project should be answered in **maximum 6 pages** (including the questions, and excluding the title page and this page, containing the instructions - the maximum total number of pages is 8).

  If you exceed the allowed space, points may be subtracted.

Part 1 [25 points]

Using the instructions provided in the project description, run memcached alone (i.e., no interference), and with each iBench source of interference (cpu, l1d, l1i, l2, llc, membw). For Part 1, you must use the following `mcperf` command, which varies the target QPS from 5000 to 55000 in increments of 5000 (and has a warm-up time of 2 seconds with the addition of `-w 2`):

```
$ ./mcperf -s MEMCACHED_IP --loadonly
$ ./mcperf -s MEMCACHED_IP -a INTERNAL_AGENT_IP  \
          --noload -T 16 -C 4 -D 4 -Q 1000 -c 4 -w 2 -t 5 \
          --scan 5000:55000:5000
```

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

(a) [**10 points**] Plot a line graph with the following stipulations:

- Queries per second (QPS) on the x-axis (the x-axis should range from 0 to 55K). (**note:** the actual achieved QPS, not the target QPS)
- 95th percentile latency on the y-axis (the y-axis should range from 0 to 8 ms).
- Label your axes.
- 7 lines, one for each configuration. Add a legend.
- State how many runs you averaged across and include error bars at each point in both dimensions.
- The readability of your plot will be part of your grade.

Answer:

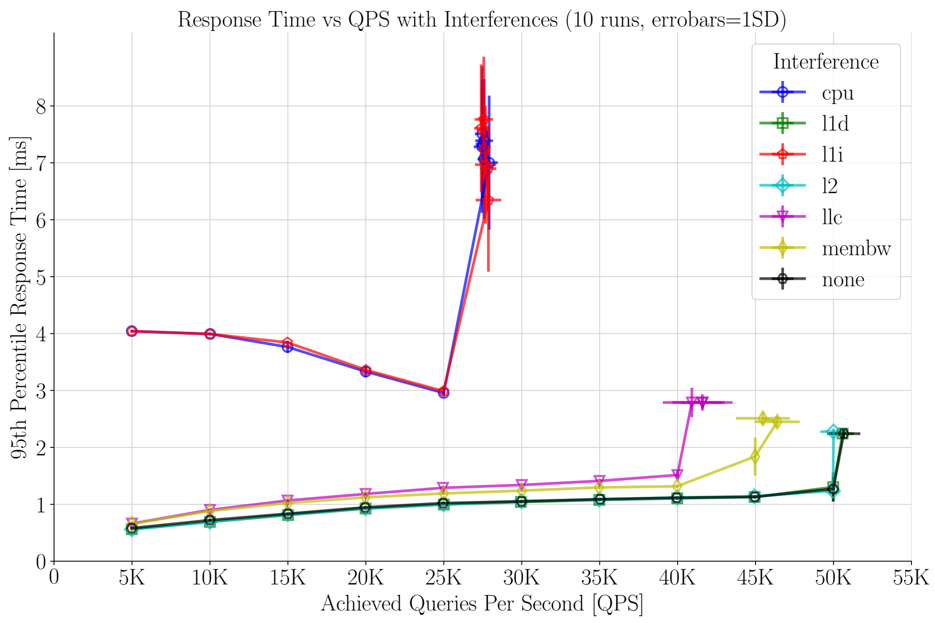

Figure 1: Average P95 response time versus achieved QPS across different interferences.

Figure 1 shows the obtained results[1]. The error bars in the plot represent the variability in the measurements across both axes. The shorter the bar the more consistent the measurements.

(b) **[6 points]** How is the tail latency and saturation point (the "knee in the curve") of memcached affected by each type of interference? What is your reasoning for these effects? Briefly describe your hypothesis.

Answer:

- `ibench-cpu`: The system's responsiveness degrades significantly under CPU interference, increasing the latency. The saturation point is at 25K.
  Explanation: This configuration substantially lowers the amount of CPU time alloted to `memcached`, meaning it has to incur in delays before it can answer to requests. However, we hypothesize that when the volume of requests grows, the system scheduler gradually assigns a greater priority to `memcached` in response to its increasing CPU demand, which approaches that of the competing `ibench` workload. `memcaded` is able to keep up until a critical point is reached at 25K queries per second (QPS), at which point the request queue starts overflowing. This happens despite the scheduler allocating up to 50%[2] of the CPU time to `memcached`.
- `ibench-l1d`: The performance of this benchmark mirrors that observed with no interference. The saturation point is at 50K.
  Explanation: `memcached` retrieves data from its key-value store, requiring memory reads. When reading from memory, the data is stored in all caches. The `ibench-l1d` benchmark causes frequent evictions from the L1 data cache. Despite these evictions, `memcached`'s response time (RT) remains unaffected, indicating its data accesses lack temporal or spatial locality, nullifying the impact of L1 cache interference.
- `ibench-l1i`: The performance behavior of this benchmark parallels that of the `ibench-cpu`, thus the previously detailed description applies. The saturation point is at 25K.
  Explanation: This behavior suggests that both `l1i` and `cpu` interferences decrease the CPU's ability to quickly execute `memcached` instructions. In this case, the CPU's L1 instruction cache is not able to retain `memcached` code as it is constantly overwritten by `ibench-l1i`. Thus, each time `memcached` is scheduled, some cycles are spent retrieving its instructions from memory, thereby increasing the RT.
- `ibench-l2`: The behavior of this benchmark is approximately identical to the one without any interference. The saturation point is at 50K.
  Explanation: It is reasonable to assume that `ibench-l2` induces numerous reads, leading to data evictions from L2. The explanation for the `l1d` interference applies here.
- `ibench-llc`: RTs slightly exceeded the baseline across most QPS levels, but the system became overwhelmed beyond the 40K saturation point.
  Explanation: The `ibench-llc` benchmark stresses the LLC with extensive reads, resulting in data evictions relevant to `memcached`. The minor increase in RT for most QPS levels suggests that `memcached`'s reads are not in the LLC. Nonetheless, the limitation seen at QPS values greater than 40K shows that `memcached` suffers from adverse effects above this threshold. This suggests that, at large request volumes, we obtain more LLC hits, thus the effects of the interference start to show.
- `ibench-membw`: RTs slightly exceeded the baseline across most QPS levels, but the system became overwhelmed beyond the 45K saturation point.

[1]The initial measurement of each configuration was discarded to mitigate startup delays, analyzing the remaining 10 out of 11 sets for each interference type.

[2]The hypothesis, grounded in Linux's fair scheduling, is confirmed by data: under `cpu` interference, `memcached`'s request handling drops to about half, indicating CPU time is split equally with `ibench`.

Explanation: This benchmark utilizes a significant part of available bandwidth, limiting the number of memory reads that can be performed in a given time-frame. Saturation occurs at 45K QPS, suggesting that `memcached`'s read operations did not initially consume the total available bandwidth.

(c) [**2 points**]

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

  Answer: The `taskset` command spawns a new program with a given affinity, which indicates the scheduler on which CPU core the process should be placed. Some `ibench` scripts run workloads designed to test the L1 and L2 caches, which are core-specific; thus, `ibench` must run on the same core as `memcached` to achieve intereference.

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

  Answer: We run some interference benchamrks on the same core as `memcached` because some CPU components such as the L1 or the execution unit are not shared across cores. Thus, the `cpu` and `l1i` interferences must be run on the same core, while the others can be run on other cores to avoid unwanted interferences.

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

Answer: Excluding `ibench-cpu` and `ibench-l1i`, the tail latency at 25K QPS consistently remained below 2 ms for all other interferences, aligning with the SLO. Thus, `ibench-l1d`, `ibench-l2`, `ibench-llc` and `ibench-membw` interferences can be collocated with `memcached` without violating the SLO.

(e) [**5 points**] In the lectures you have seen queueing theory.

- Is the project experiment above an open system or a closed system? Explain why.

  Answer: It is a closed system because the number of clients is fixed and each client waits for the response to its previous query before sending a new one[3]. Thus, it can also be said that the system is self-adjusting.

- What is the number of clients in the system? How did you find this number?

  Answer: The number of clients is 64[4]. This follows from the used flags[5], which configure the system with a single Measure Client running `mcperf` with 16 threads, each maintaining 4 connections. Each connection is treated as an individual client.

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

  Answer: Figure 2 provides an overview of the system with 64 clients sending requests. The client's requests to `memcached` are queued, processed, and then the result is sent back. The clients wait for a response before sending a new request.

[3]From `mcperf` GitHub description: "it waits for a reply before sending the next request".

[4]This counts only the connections opened by the Measure Client, which are used during the benchmark. We are aware that before running the benchmark the Control Agent connects to memcached to fill up the key value store, but we chose to ignore this as this performance is not measured.

[5]A detailed description of the `mcperf` flags can be found here: cmdline.ggo on GitHub

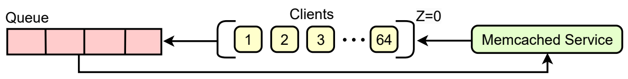

Figure 2: Sketch of the closed system.

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

  Answer: The formula to compute the average RT for a closed system is:

  $$R = \frac{N}{X} - Z = \frac{64}{X} - 0$$

  In this case, R (P95 RT) is given by plugging $N = 64$, $Z = 0$ and X as the desired QPS.

Part 2 [31 points]

1. **Interference behavior [20 points]**

   (a) **[10.5 points]** Fill in the following table with the normalized execution time of each batch job with each source of interference. The execution time should be normalized to the job's execution time with no interference. Round the normalized execution time to 2 decimal places. Color-code each cell in the table as follows: **green** if the normalized execution time is less than or equal to 1.3, **orange** if the normalized execution time is over 1.3 and less or equal to 2, and **red** if the normalized execution time is greater than 2. The coloring of the first three cells is given as an example of how to use the cell coloring command. **Do not change the structure of the table. Only input the values inside the cells and color the cells properly.**

Workload	`none`	`cpu`	`l1d`	`l1i`	`l2`	`llc`	`memBW`
blackscholes	1.00	1.27	1.38	1.57	1.37	1.49	1.39
canneal	1.00	1.53	1.56	1.60	1.54	2.23	1.61
dedup	1.00	1.67	1.24	1.91	1.21	2.09	1.64
ferret	1.00	2.03	1.60	2.43	1.57	2.82	2.37
freqmine	1.00	1.99	1.03	2.60	1.04	1.85	1.56
radix	1.00	1.01	1.12	1.16	1.09	1.65	1.10
vips	1.00	1.60	1.66	1.75	1.71	1.96	1.58

   (b) **[7 points]** Explain what the interference profile table tells you about the resource requirements for each job. Give your reasoning behind these resource requirements based on the functionality of each job.

   Answer:

   - `blackscholes:` Results show that the code is highly effected by cache and memory bandwidth saturation. `l1i` interference causes the largest slowdown.
     Explanation: This benchmark is comprised of many spatially local instructions to solve differential equations, making it susceptible to the `l1i` interference. The benchmark is also sensitive to `llc`, which adds additional latency to memory reads.
   - `canneal:` The benchmark under-performs with any interference, especially during LLC cache saturation, indicating high CPU and RAM usage.

Explanation: This benchmark focuses on concurrent algorithms and memory access patterns, which explains its sensitivity to CPU and RAM-targeted interferences. The large number of reads required, in the face of LLC cache evictions, undermines performance.

- `dedup:` The benchmark's performance significantly declines with `cpu`, `l1i`, `memBW`, and `llc`.

  Explanation: The behavior aligns with `dedup`'s nature, which demands intensive CPU and RAM usage. Testing revealed notable slowdowns under CPU and L1 cache stress, but LLC saturation is the most noticeable, due to the amount of memory reads required to load the large dataset.

- `ferret:` This benchmark behaves similarly to `dedup`, showing remarked slowdowns under `cpu` and `memBW` interferences.

  Explanation: Considerations for `dedup` also apply to `ferret`, which is even more susceptible to interferences on instruction execution. Further, this benchmark focuses on similarity searches in large datasets, requiring intensive CPU and memory usage. Thereby, it suffers heavily under `cpu` and `memBW` interferences.

- `freqmine:` It shows significant slowdown under `cpu`, `l1i`, and `llc` interferences, highlighting its CPU and RAM intensity.

  Explanation: This benchmark stresses parallel computations and memory access patterns. Performance drops when instruction execution is hindered by `l1i` cache misses or CPU load. The impact of LLC cache filling is also notable, as it forces reliance on slower main memory for data, further increasing latencies.

- `radix:` This benchmark is resilient to most interferences, with only LLC cache saturation impacting its performance.

  Explanation: This benchmark, uses a parallel radix sort algorithm. It is thus memory-bound, and unaffected by CPU and `l1i` interferences. Its resilience to L1 and L2 data cache saturation also suggests that it deals with a large dataset, where performance is only impacted by LLC saturation.

- `vips:` This benchmark is affected by all interferences, indicating it is not exclusively CPU or memory-bound, with a slightly greater sensitivity to LLC saturation.

  Explanation: This benchmark, focused on image processing, demands significant CPU and memory resources, thereby all interferences highly impact it. The most significant interference is the LLC saturation, which is coherent given the amount of data required to do image processing.

(c) [**2.5 points**] Which jobs (if any) seem like good candidates to collocate with memcached from Part 1, without violating the SLO of 2 ms P95 latency at 40K QPS? Explain why.

Answer: Figure 1 shows that only `l1i` and `cpu` interferences breach the 2ms SLO at P95, suggesting benchmarks like `radix`, minimally impacted by these, are fit for collocation with `memcached`. `blackscholes` might also be a candidate, but poses a risk due to its notable sensitivity to `l1i` and `cpu`.

2. **Parallel behavior [11 points]**

(a) [**7.5 points**] Plot a line graph with speedup as the y-axis (normalized time to the single thread config, Time_1 / Time_n) vs. number of threads on the x-axis (1, 2, 4 and 8 threads

- see the project description for more details). Pay attention to the readability of your graph, it will be a part of your grade.

Answer:

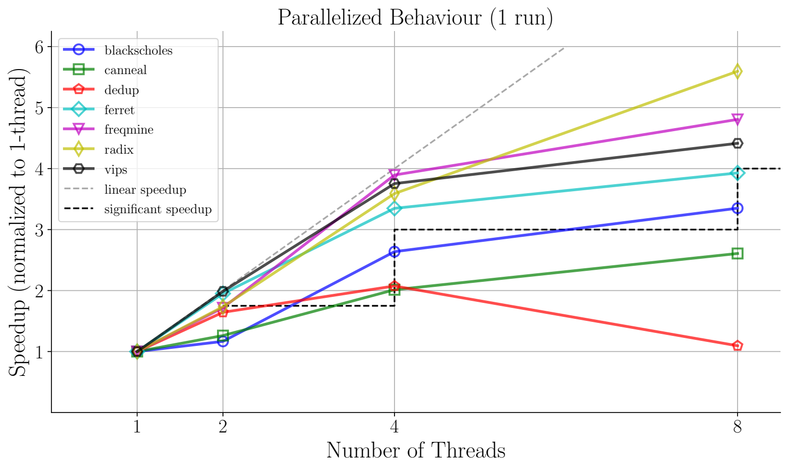

Figure 3: Sweep of parsec benchmarks across different thread counts.

(b) [**3.5 points**] Briefly discuss the scalability of each job: e.g., linear/sub-linear/super-linear. Do the applications gain significant speedup with the increased number of threads? Explain what you consider to be "significant".

Answer: Significant speedup, modeled as a step function, sets thresholds: 1.75x for 2 threads, 3x for 4 threads, and 4x for 8 threads. A benchmark's speedup is considered linear if its interpolation approaches a $y = \alpha x$ line (with $0 < \alpha \leq 1$).

- `blackscholes`: The job has a sub-linear scaling, with a marked speedup observed from 2 to 4 threads. It fails to meet our criteria for significant speedup.
- `canneal`: The job has a moderate linear scaling, with a mild speedup throughout the sweep. It falls short of our criteria for significant speedup.
- `dedup`: The job's sub-linear scaling noticeably under-performs the ideal speedup, with a high and moderate speedup observed from 1 to 2 threads and from 2 to 4 threads respectively. From 4 to 8 threads it intriguingly exhibits a parallel slowdown, likely stemming from synchronization overheads. It fails to meet our criteria for significant speedup.
- `ferret`: The job shows a sub-linear scaling. Despite this, it satisfies our criteria for significant speedup, albeit marginally in the 8-thread configuration.
- `freqmine`: While the job shows substantial speedup from 1 to 2 threads and a nearly linear speedup from 2 to 4, its overall performance is not deemed linear due to diminished gains from 4 to 8 threads. It meets our criteria for significant speedups, albeit marginally in the 2-thread configuration.
- `radix`: This benchmark exhibits linear scaling throughout the sweep range with a significant speedup. We also noticed a diminishing return in performance when increasing the number of threads.
- `vips`: While the job shows substantial speedup from 1 to 2 threads and a nearly linear speedup from 2 to 4, its overall performance is not deemed linear due to diminished gains from 4 to 8 threads. We notice a diminishing return in performance when increasing the number of threads.