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
April 24, 2023

Instructions

- Please do not modify the template, except for putting your solutions, group number, names and legi-NR.
- Parts 1 and 2 should be answered in maximum six pages (including the questions).
  If you exceed the 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 30000 to 110000 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 30000:110000:5000
```

Repeat the run for each of the 7 configurations (without interference, and the 6 interference types) **at least three times** (three 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.

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

- Queries per second (QPS) on the x-axis (the x-axis should range from 0 to 110K). (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.

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

(c) [**2 points**] Explain the use of the `taskset` command in the container commands for memcached and iBench in the provided scripts. Why do we run some of the iBench benchmarks on the same core as memcached and others on a different core?

(d) [**2 points**] Assuming a service level objective (SLO) for memcached of up to 1.5 ms 95th percentile latency at 65K QPS, which iBench source of interference can safely be collocated with memcached without violating this SLO? Briefly explain your reasoning.

(e) [**5 points**] In the lectures you have seen queueing theory. Is the project experiment above an open system or a closed system? What is the number of clients in the system? Sketch a diagram of the queueing system and provide an expression for the average response time. Explain each term in the response time expression.

Part 2 [30 points]

1. **Interference behavior [19 points]**

   (a) **[11 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 field 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 up to 2, and **red** if the normalized execution time is greater than 2. Briefly summarize in a paragraph the resource interference sensitivity of each batch job.

Workload	`none`	`cpu`	`l1d`	`l1i`	`l2`	`llc`	`memBW`
blackscholes	1.00						
canneal	1.00						
dedup	1.00						
ferret	1.00						
freqmine	1.00						
radix	1.00						
vips	1.00						

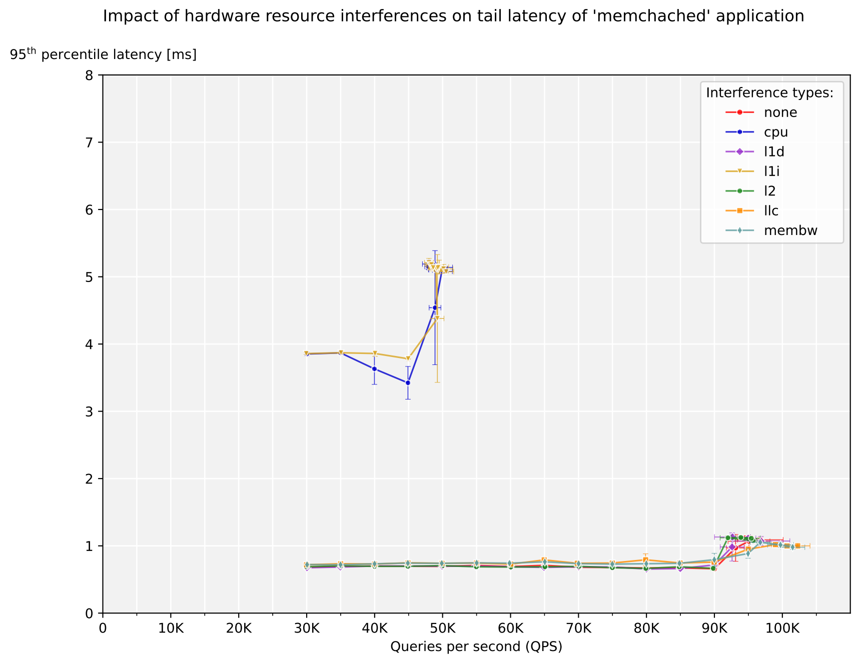
   (b) **[8 points]** Explain what the interference profile table tells you about the resource requirements for each application. Which jobs (if any) seem like good candidates to collocate with memcached from Part 1, without violating the SLO of 2 ms P95 latency at 40K QPS?

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

   Plot a single 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). Briefly discuss the scalability of each application: 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".

Part 1 [Answers]

(a) Each of the seven configurations displayed in the plot above was run three times. While the data points represent the mean values accross the three runs, the error bars depict the standard deviation in both dimensions.



(b) The plot shows changes in the 95th percentile tail latency and saturation points of the *memcached* application when deployed on the same VM as various interference microbenchmarks. Notably, there are two main patterns that emerge:

- The microbenchmarks that provoke L1 data cache (`l1d`), L2 cache (`l2`), last-level cache (`llc`) or memory bandwidth (`membw`) interference result in the lines clustered towards the bottom of the plot. As can be seen, the 95th percentile tail latencies caused by those interference types are very close to the measurements taken without additional hardware interference (`none`). In addition to this, the saturation points ("knee in the curve") occur rather late, at around $90K$ queries per second (QPS) and the maximal achievable workload levels off at around $95K$-$100K$ QPS with a tail latency of around 1 ms.
- The `ibench-cpu` and `ibench-l1i` (L1 instruction cache) microbenchmarks incur a stronger performance penalty on the application than the previously mentioned microbenchmarks. The saturation point of *memcached* for `cpu` and `l1i` interference occurs a lot earlier at around $45K$ QPS. The maximal achievable workload and the corresponding tail latency stagnate at around $50K$ QPS and 5.1 ms respectively.

From the sudden increase in response time at specific query rates, one can infer that the increasing load on the system causes jobs to queue up at the server and the system has therefore hit its saturation point. While there are clearly visible differences in how much load each configuration is able to handle, it is important to note that all setups eventually fail to achieve the target QPS as soon as jobs start to spend more time in the queue. Overall, measurements start to vary significantly once the saturation point of each configuration is reached. The fluctuation in measurements is indicated by the error bars displayed in the plot.

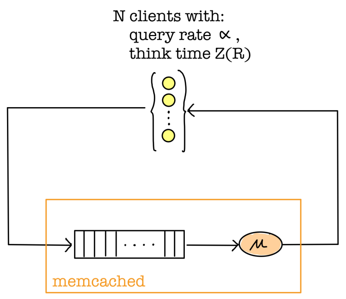
Upon inspection of the data we gathered, we can conclude that the *memcached* application is more susceptible to interference with loading and executing instructions than to interference with loading and storing data. Our hypothesis to explain the observed behaviour is as follows: The L1 data cache (`l1d`), L2 cache (`l2`), last-level cache (`llc`) and memory bandwidth (`membw`) interference show almost no negative impact on the performance of *memcached*. We assume that since the application is processing numerous queries from various clients, even without additional interference on the data cache level, *memcached* frequently experiences cache misses as data is often only accessed once. Therefore, the impact of adding data cache interference and hence causing data that was not recently accessed to be evicted from cache is hardly visible. We assume that `l1i` and `cpu` interference increase the tail latency of the application so drastically because instruction misses require the CPU to fetch instructions from memory more often. The CPU is responsible for executing instructions and if it is kept busy with fetching instructions or executing tasks for another application, this results in a performance degradation of our measured application.

(c) The `taskset` command is used to set the CPU affinity for a process [1], to ensure that a process is only run on a specific core(s). In our experiments, all interference microbenchmarks besides `ibench-llc` and `ibench-membw` ran on the same core as the *memcached* application (core 0). The reason for running the `llc` and `membw` on a different core (core 1) is to reduce additional effects on other resources. As the last-level cache and the memory are shared by all cores, it is also possible to study the effect of hardware interference by running these microbenchmarks on another core. In contrast, the L1 and L2 caches are not shared amongst cores, so in order for this interference to be studied it had to be running on the same core as *memcached*.

(d) Given an SLO of 1.5ms 95th percentile tail latencies at 65K QPS, interference on the L1d, L2, LLC and memBW can safely be collocated with memcached. The plot shows that at 65K, interference on these resources does not cause the tail latencies to grow above 0.7ms. The operation point of 65K QPS is at around 70% of the load at the saturation point.

(e) From our understanding of the *mcperf* load generator tool, given a targeted number of queries (QPS_{target}) all clients query the *memcached* server at a rate related to QPS_{target}. Lets define the query rate per client to be $\alpha = \frac{QPS_{target}}{N}$ queries per second, with N being the total number of clients. Clients wait to receive a response from the server and only then send their subsequent request. Therefore we model the experiment setup as an interactive, closed system with adaptive think time Z and queue size N. We define Z as follows: $Z = \frac{1}{\alpha} - R$, if $R < \frac{1}{\alpha}$, else $Z = 0$. Here, R represents the response time of *memcached* in seconds and $\frac{1}{\alpha}$ corresponds to the minimum time (seconds) between two queries of a client to reach QPS_{target}.
$N = 4$ connections * 16 threads $= 64$ clients.

Response time R (with throughput = service time = μ):
a) underload → jobs do not yet queue up: $R = \frac{1}{\mu}$
b) stationary → queue is full (N jobs in queue, Z = 0): $R = \frac{N}{\mu}$



Part 2 [Answers]

1. **Interference behavior**

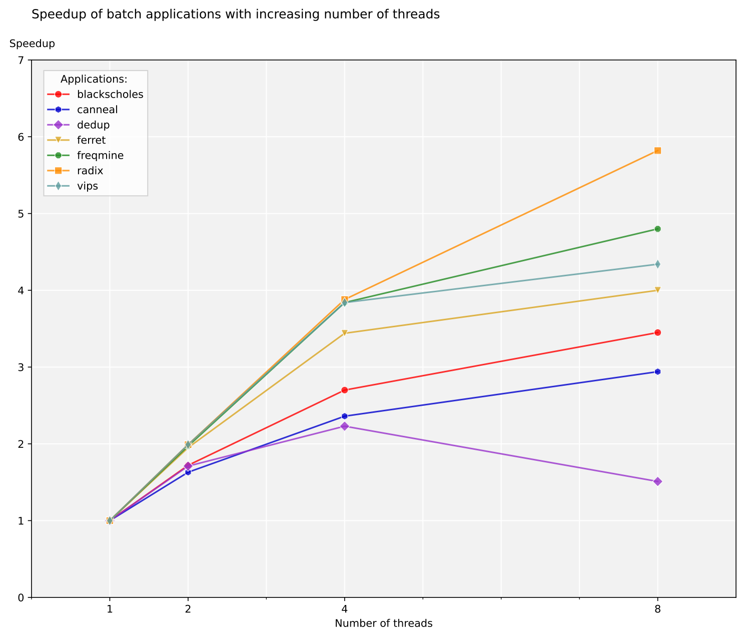
   (a)

Workload	none	cpu	l1d	l1i	l2	llc	memBW
blackscholes	1.00	1.29	1.22	1.58	1.20	1.55	1.33
canneal	1.00	1.17	1.16	1.32	1.26	1.99	1.48
dedup	1.00	1.65	1.16	2.05	1.17	2.22	1.53
ferret	1.00	2.35	1.45	2.99	1.05	2.72	2.04
freqmine	1.00	2.12	1.06	2.14	1.07	1.92	1.56
radix	1.00	1.06	1.00	1.04	1.01	1.60	1.03
vips	1.00	1.72	1.53	1.93	1.53	1.89	1.70

The `real` [2] output in the PARSEC-benchmark suite was used for the table. As a general trend, the workloads were most sensitive to interference on the L1 instruction cache, last-level cache and on the memory bus. The interference on the memory bus causes a decrease in available BW and longer latency for memory accesses. Interference on the last-level cache causes thrashing which would require frequent main memory accesses. This is particularly problematic because the data-movement between main-memory and cache is an order of magnitude slower than data-movement between different layers of cache, hence why interference on LLC affects all workloads so negatively. Conversely, due to data-movement between cache levels not being as expensive, most workloads were not affected very negatively by interference on the L1d and L2. The `radix` workload is not sensitive to interference on any other resource, except the LLC. Of the remaining workloads, `blackscholes` and `canneal` are the least sensitive to interference on the CPU. Both workloads are most sensitive to interference on the L1i, LLC and memBW. This is especially true for `canneal`. `ferret` and `freqmine` are the workloads which are most sensitive to interference on the CPU. They also showed significant performance degradation when interference was running on L1i. `ferret` also experienced significant slow-down when interference was run on LLC and memBW. The `dedup` workload suffered mostly when interference was run on the CPU, L1i, LLC and memBW. The `vips` workload showed a medium level of performance degradation for all kinds of interference.

   (b) `blackscholes` and `canneal` do not suffer significant (red-region) performance degradation for any of the interference profiles. They experience their highest degree of performance degradation with interference on L1i, LLC and memBW. This means that they are memory intensive workloads and not very computationally expensive, since there is almost no degradation for CPU interference. This was confirmed in [3] where it can be seen that they have the least amount of instructions, excluding `radix`. `dedup` experienced significant slow-down when interference was run on L1i and LLC. It also experienced a lesser slow-down with interference on the CPU. This means that this workload is mostly memory intensive. `ferret` experienced a high degree of performance degradation across all interference profiles meaning that it is resource intensive - both in terms of computation and memory. It was also significantly effected by interference on the L1i. This could mean that typically, the working set of instructions is able to fit inside L1i cache or that there are not that many cache misses. This interference on L1i causes a lot of misses and accesses to slower levels of cache. `freqmine` experienced a high degree of performance degradation with interference on CPU and L1i (same reason as previous workload), and to a lesser extent with interference on LLC and memBW. This workload is also resource intensive, particularly computationally. `vips` experienced a similar level of slow-down for all of the interference profiles which were run however, none of them were significant. This meant that this workload was not too resource intensive. It also meant that the workload is balanced between computation and memory requirements. Finally, `radix` experienced almost no slow-down for all interference profiles except LLC. It is not resource intensive at all. Experiencing almost no slow-down on L1i could mean that misses occur frequently in this cache even without interference [4]. The plot in Part 1 showed that *memcached* experienced the highest increase in tail-latencies when interference was run on the CPU and L1i. Due to this, workloads which were intensive in these areas were immediately excluded. `vips` was also excluded because it showed a slow-down across all interference profiles.`radix` seems to be the best candidate to be col-located with *memcached* without violating the SLO. This is because it suffered very little performance degradation when interference was run on CPU and L1i. `Blackscholes` and `canneal` could also be potential candidates as they are not sensitive to interference on CPU but have a medium-level of sensitivity to L1i cache interference.

2. **Parallel Behaviour**



The above plot shows that `blackscholes`, `canneal` and `dedup` scaled sub-linearly at a similar rate up until 2 threads. Beyond this, `canneal` scaled slightly worse than `blackscholes`. `dedup` scales worse to 4 threads and its performance worsens when using 8 threads. `ferret` scales up linearly until two threads, and scales better than the aforementioned 3 workloads but worse than `radix`, `freqmine` and `vips`. The latter 3 workloads scale almost linearly up until 4 threads, and beyond that `radix` scales best of all workloads. None of the workloads scaled in super-linearly. We consider a speedup to be ”significant” if it is at least 1.5x faster when doubling the number of threads. Going from 1 to 2 threads results in a significant speedup for all workloads. Increasing the number of threads from 2 to 4 still results in a significant speedup for all the workloads except for `canneal` and `dedup`. However going from 4 to 8 threads only results in a significant speedup for the `radix` workload.

References

[1] https://man7.org/linux/man-pages/man1/taskset.1.html

[2] https://stackoverflow.com/questions/556405/what-do-real-user-and-sys-mean-in-the-output-of-time1

[3] https://parsec.cs.princeton.edu/doc/parsec-report.pdf

[4] https://pages.cs.wisc.edu/ markhill/restricted/isca95_splash2.pdf