Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Systems@ETH Zürich

Cloud Computing Architecture

Semester project report

Group 39
Peter Abdel Massih - 20-810-214
Leonardo Favento - 22-953-533
Sofija Kotarac - 22-907-331

Systems Group
Department of Computer Science
ETH Zurich
April 6, 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.

See Figure 1.

(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.

With no interference the 95th percentile latency is almost constant at around 0.7 ms and the saturation point is at 95K queries per second.
The CPU interference and the L1I interference are the ones that affect the tail latency and the saturation point the most. The tail latency starts more than 4x bigger than with no interference and the saturation point comes at around 50K queries per second, where the latency is more than 5x bigger than with no interference. This is due to the fact that the retrieval of data from memcached is an operation cpu-intensive, and the L1I cache has a strong impact on the performance: while the processor fetches an instruction, it usually sits idle. In fact, l1 instruction cache is used to demarcate individual instructions to improve decode speed, showing that instructions to fetch data are quite similar.
L1D, L2, LLC, and membw interference have no impact on the tail latency below 75K queries per second. After that saturation point, membw and llc interference cause the latency to increase slowly reaching 1ms at 95k QPS. On the other hand, l2 and l1d lead to a latency of 1 ms after 90k. In the first period, the latencies are constant for each test case, which might be the cause of the number of hits and misses being similar to the no interference case.

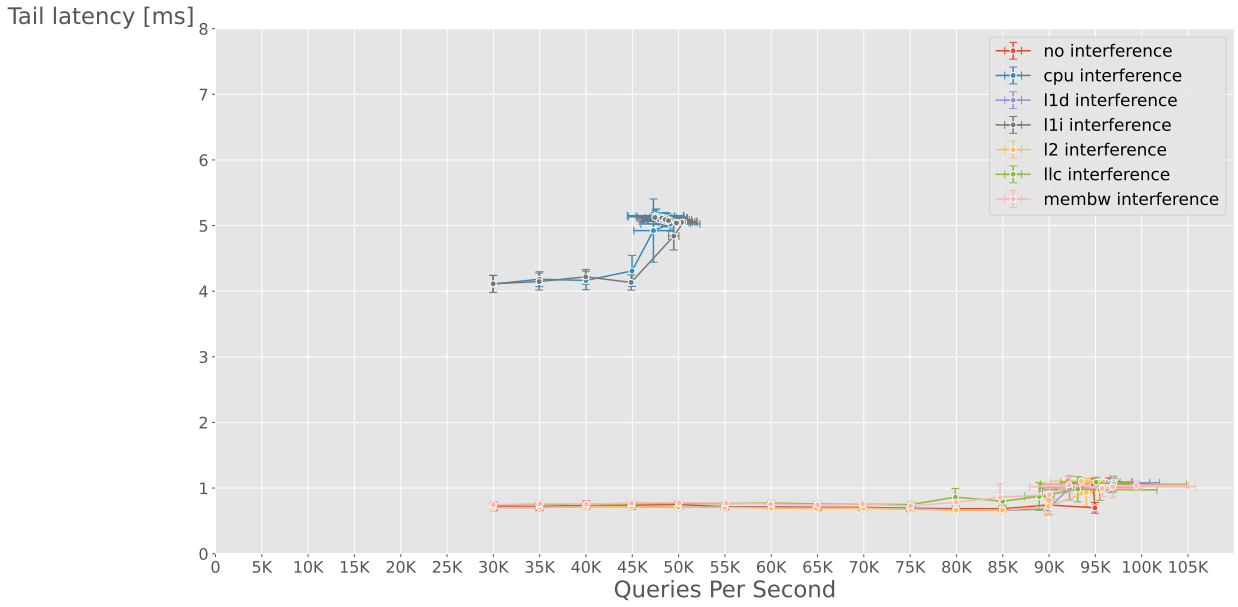


Figure 1: Plot of the 95th percentile latency depending on the achieved queries per second. The points in the graph are the mean of values obtained with 3 runs for each configuration. The error bars represent the standard deviation, so they tell us how much the actual measurements vary from the mean.

Whereas, after the saturation with llc, the chance of missing in the cache increases - more QPS cause higher latency. One explanation would be that the values accessed through the keys in memcached are sparse at first with very low repetition of the same data accessing. With more QPS comes more data accesses, same data accesses happen more often which adds delay. Thus, the missing probability becomes higher than the no interference case.
In the case of membw, the latency isn't greatly affected, just like with llc. The reason is that memcached operates on data of small size, so the memory bandwidth required is low. In the second period (after 75K QPS), memory bus congestion causes throughput to become slightly slower.
For l2 and l1d test cases , as stated earlier in the llc case the same piece of data doesn't seem to be accessed often enough, thus the chunk of data gets evicted even when we have no interference. After the knee of the curve (90k QPS), the latency increases lightly, this might be because with high QPS some repeated instruction set (in the l2 instruction cache), or some recently accessed data (l1d) might be ejected due to interference.

(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?

The use of `taskset` command is to schedule programs to specific CPU cores. In the *memcached* and *iBench* scripts, for example *taskset -c 0 ./cpu 1200* launches the program ./cpu on the 0^{th} core of the processor whilst passing the arguments to the program, in this case 1200.

By observing the scripts, we found that memcached runs on the 0^{th} core, and that the following iBench benchmarks are also launched on the same core: ibench-cpu, ibench-l1d, ibench-l1i, ibench-l2. This is because we want to interfere with the cpu core that runs memcached so the cpu interference should be on the same core. Similarly, L1d, L1i and L2 are core specific caches, meaning

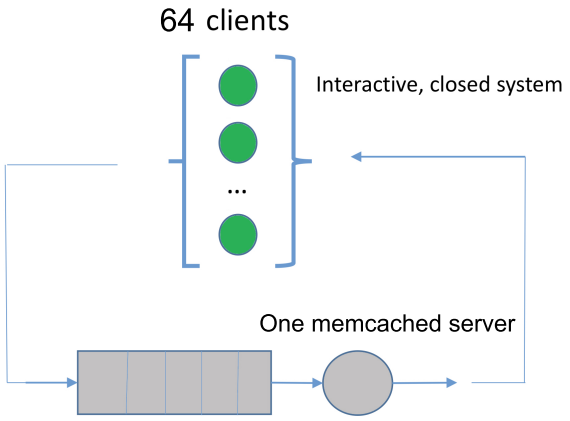
each core has its own specific L1 and L2 cache. Thus if we want to interfere with memcached on L1d, L1i and L2 we must run the interference on its same core. Whereas ibench-llc and ibench-membw run on the 1^{st} core, because LLC and remote memory are shared among all cores.

(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.

At 65K queries per second l1d, l2, llc and membw interferences don't affect the 95th percentile latency as the experimental results show. Therefore, since all these interferences keep the 95th percentile latency at 0.7 ms (like in the case without interference), all these iBench sources of interference can be collocated without any effect that could violate the SLO of 1.5 ms.

(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.

In our case, the project experiment is a closed system since the load to the memcached database comes from a limited set of clients. Our system thus tends to stability and our client waits for responses before sending next requests. There is 64 clients in the system sending queries, shown by the flags "-T 16 -A" 16 threads in the client agent each holding "-c 4" connections, thus $4 * 16 = 64$ connections, all handled by the unique thread of the memcached server "-t 1" in the yaml file. The think time is zero ($Z = 0$) for the client, it just waits for a reply before sending the next request. The throughput is the queries per second achieved at the saturation point, where the throughput plateau, and the response time now increases. The diagram and response time expression is then:



Based on Little's Law (or the interactive Law for closed Systems): average response time $\bar{R} = \frac{N}{X} - Z$ where $N = 64$ is the number of clients, X is the throughput (QPS at saturation), and Z is the thinking time (here 0 since it just waits for the reply before sending another request) $\implies \bar{R} = \frac{64}{X}$. This expression grants accurate results when the system is not subject to interference, or when the interference doesn't strongly affect the system. For instance, in the no interference case, the saturation point comes around 95k QPS thus the theoretical response time becomes $\bar{R} = \frac{64}{95000} \approx 0.67ms$ against an experimental result of $0.69ms$. When the interference stresses significantly the memcached server, we experience an offset between the theoretical and experimental values due to the additional delay introduced.

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	1.31	1.16	1.67	1.18	1.48	1.41
canneal	1.00	1.11	1.23	1.33	1.20	1.86	1.28
dedup	1.00	1.72	1.42	2.12	1.24	2.28	1.73
ferret	1.00	1.86	1.06	2.35	1.12	2.58	2.04
freqmine	1.00	2.02	1.04	2.04	1.03	1.83	1.59
radix	1.00	1.07	1.11	1.20	1.13	1.65	1.17
vips	1.00	1.68	1.57	1.95	2.12	1.91	1.70

   Blackscholes application is not very sensitive to any interference. Its performance is affected mostly by interference on L1 instruction cache, last level cache, memBW and, to lesser extent, CPU.
   Canneal and radix are also not very sensitive to interference, except for interference on last level cache, which causes the programs to slow down significantly.
   Dedup suffers greatly from interference. The most critical interferences are the ones on L1 instruction cache and on last level cache, which cause the execution to take more than twice the time it takes without any interference. CPU and memBW also have a significant impact on the performance of the program, while L1 data cache and L2 cache interferences only have a minor influence.
   Ferret and freqmine have similar sensitivity to interference: both of them suffer greatly from CPU, L1 instruction cache, last level cache and memBW interferences. Hence, they are both CPU- and data-intensive programs, although the experimental results show that ferret is the more sensitive to interference on data (llc and memBW), and freqmine suffers more from CPU interference.
   Vips performance decreases significantly with all the tested sources of interference. The one that affects vips the most is L2 cache interference, followed by L1 instruction cache and last level cache.

   (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?

   All jobs showed sensitivity to interference on L1 instruction cache, since instructions are repeated frequently, and on last level cache due to the need to directly access memory, which takes more time than accessing the next level of cache.
   Blackscholes : as shown in the previous answer, blackscholes is not a resource intensive application, but it is mostly affected by l1i, llc, membw and cpu. It slows down less when the CPU is stressed compared to the memory, therefore it's slightly data intensive.

Canneal and radix : these are also not resource intensive application, but they are more leaning towards data-intensive programs as shown by their last level cache memory dependency.
Dedup and Ferret suffer from last level cache, memBW and l1i interference, which suggests that they require access to a lot of data (ferret is the most data-intensive tested workload). However, their performances are not greatly affected by L1 data cache and L2 cache interferences, likely because there would be many misses on those levels of cache even without interference. Additionally, the programs slow down significantly if the CPU is stressed, indicating that they are also quite cpu-intensive.
Freqmine: its most impactful sources of interference are similar to the ones of dedup and ferret, with the difference that the major negative consequences on its performance are due to CPU interferences. Freqmine is in fact the most cpu-intensive among the tested workloads, although it suffers greatly also from interference on last level cache and memBW, which shows that it also requires a good amount of memory accesses.
Vips performance suffers from all the sources of interferences, in particular by L2 cache, followed by L1 instruction cache and last level cache. The strong impact of L2 cache means that the program needs to access contiguous data, which gets replaced by the effect of the interference, thus causing repeated cache misses and a decrease in performance.

As shown in part 1, the performance of memcached at 40K queries per second is affected only by interference on CPU and L1 instruction cache, while all other interferences don't cause any performance loss. Radix and canneal workloads are the best ones to be collocated with memcached without violating the SLO of 2 ms P95 latency, while Blackscholes stresses moderately the CPU and the L1i, and since we don't know precisely the magnitude of its impact, we can't be sure if it can be collocated with memcached or not. Dedup, ferret, freqmine and vips all use more extensively the two critical resources, making them not good candidates.

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".

Figure 2 shows the results obtained running each application 3 times for each number of threads configuration (1, 2, 4 and 8 threads). As Amdahl's Law explains, speedup is limited by the fraction of the program that can be accelerated, therefore there is always a percentage of the program that can be parallelized while the rest of the program can't be run in parallel, so different applications show different speedups while testing them over different number of threads.

Radix, freqmine and vips get the most relevant speedup: their scalability is almost linear up to 4 threads. The three applications have sub-linear scalability when raising the amount of thread to 8, with radix being the one which gains the most out of it having a speedup of slightly more than 50%, while freqmine and vips only have a performance improvement of less than 25%.
Ferret has linear scalability when increasing from 1 to 2 threads, and the observed speedup from 2 to 4 threads is still significant, although sub-linear, improving the runtime of around

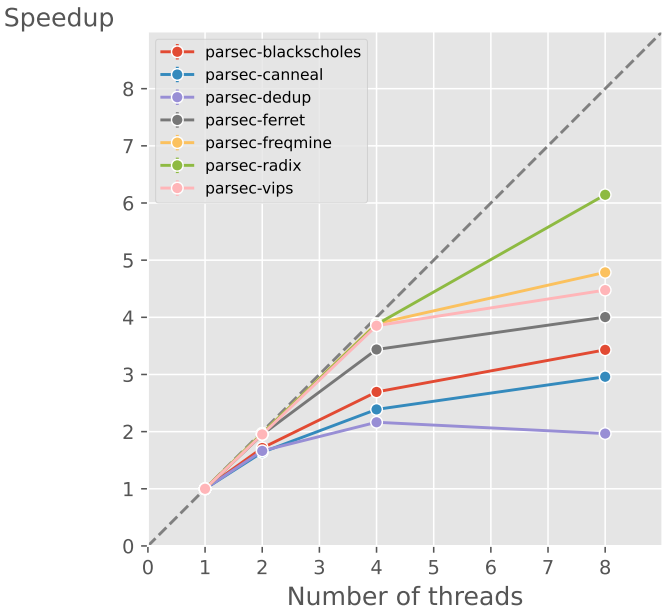


Figure 2: Speedup of each application depending on the number of threads on which it is executed. The values were obtained by averaging 3 runtimes for each workload in each thread configuration.

75%. The program performance though, doesn't register relevant improvement when going from 4 to 8 threads, having a speedup of around 15%.

Blackscholes and canneal have sub-linear scalability even when increasing the number of threads from 1 to 2 but the performance improvement is still significant, being around around 70%. When going from 2 to 4 threads, their performance improvements are still valuable, with blackscholes having a speedup just below 60%, and canneal around 45%. Raising the number of threads from 4 to 8 is not very profitable, making the program run less than 25% faster.

Dedup gets the least performance improvement by augmenting the number of threads: the speedup increasing from 1 to 2 threads is significant, being almost the same as the one of blackscholes and canneal, around 70%. When upgrading to 4 threads, the registered improvement is of 30%, and increasing the number of threads to 8 is detrimental, having the program's runtime decreased by 10%. This shows how in some applications, parallelizing excessively can have deleterious consequences due to the synchronisation overhead overshadowing the parallelization benefits.

All in all, dedup shows performance improvement by increasing the number of threads on which it's executed up to 4, while all the other tested applications improve their runtime up to 8 threads. Considering ideal a linear correlation between the increased number of threads and the speedup (doubling the amount of threads halves the execution time), we consider 'significant' a speedup of 50% when doubling the amount of threads so that the ratio between the cost of increasing the number of threads and the obtained speedup is not very high. Thus, significant speedups are obtained for canneal and dedup when increasing the number of threads up to 2, for blackscholes, ferret, freqmine and vips when the threads are upgraded up to 4, and for radix the improvements are significant up to 8 threads.