Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Systems@ETH Zürich

Cloud Computing Architecture

Semester project report

Group 49
Kehong Liu - 22943385
Yuening Yang - 21947502
Minjing Shi - 22943971

Systems Group
Department of Computer Science
ETH Zurich
July 20, 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]

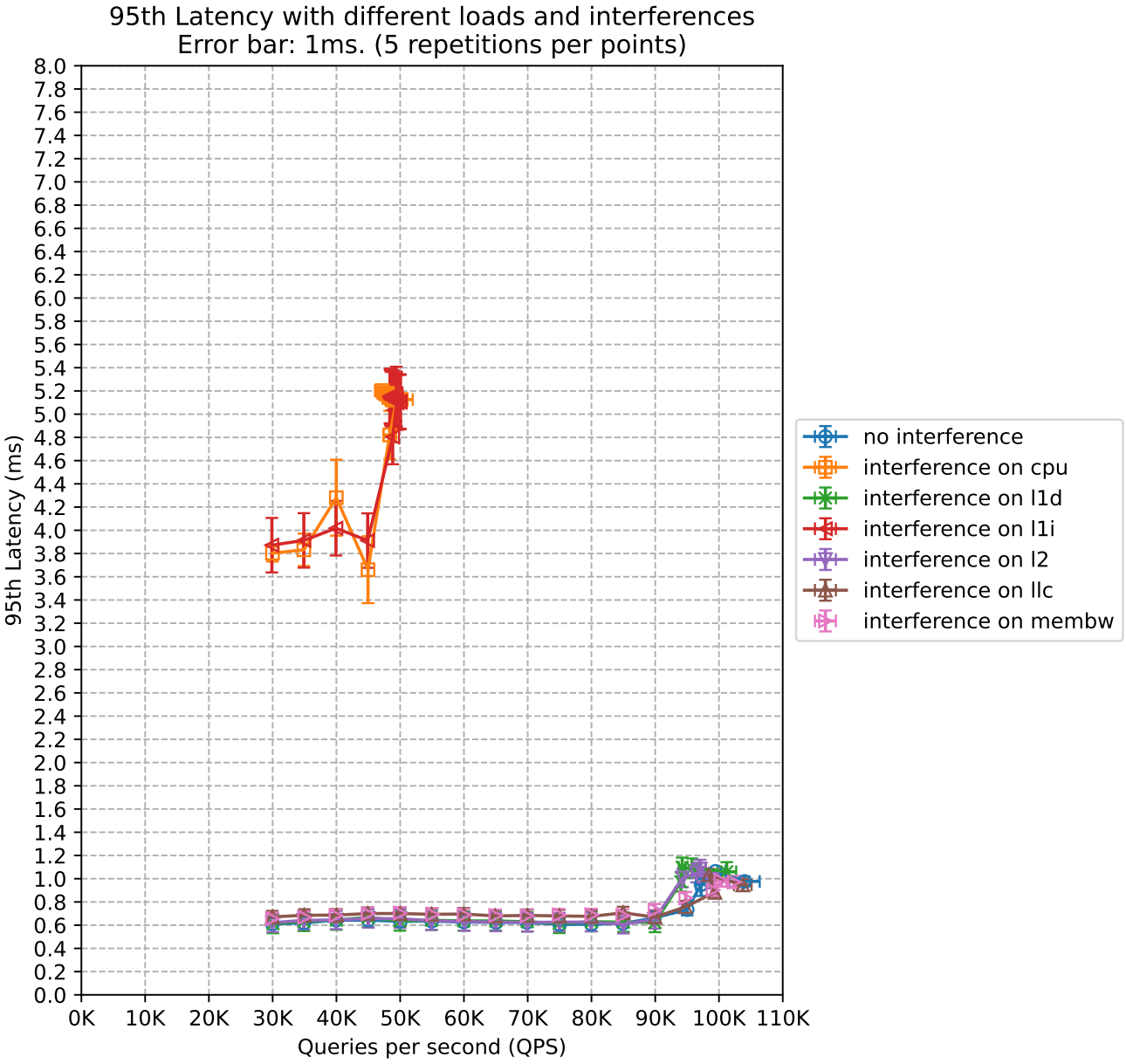
Using the instructions provided in the project description, run memcached alone (i.e., no interference), and with each iBench source of interference (cpu, l1d, l1i, l2, llc, membw). For Part 1, you must use the following `mcperf` command, which varies the target QPS from 30000 to 110000 in increments 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.

The tail latency saturation point for no interference and L1D, L2, LLC, memBW interference all appear around 90K QPS. However, L1D and L2 interference leads to a steeper increase in latency while the latency for no interference, LLC, and memBW are roughly the same. On the other hand, the tail latency saturation point for CPU and L1I interference appears much earlier at 45K QPS, and memcached were never able to achieve more than 50K QPS.

Our hypothesis for such behavior is that memcached is compute bound, i.e. the latency is determined by the speed of computation rather than the speed of data transfer. One possible reason for it to be compute bound is that the dataset queried has some locality and generally fits in upper cache levels, minimizing the time spent on transferring data and spending the majority of the time on computation. Thus, when there's no interference on CPU resources, the latency only gets

saturated when QPS is large and much more data needs to be transferred. The steeper increase in latency under L1D and L2 interference is possibly due to the locality of the data. As mentioned previously, it is likely the dataset queried fits in the upper levels of the cache when there's no interference. Thus when interference on L1D and L2 is present, the number of cache misses increases, memcached now need to spend more time transferring data from lower memory hierarchies, leading to a steeper increase in latency. On the other hand, since the dataset fits in L1D and L2 cache when there's no interference, there will be minimal access to LLC and memory to begin with, therefore interference on LLC and memBW has little to no effect on the latency. When there's interference on CPU and L1I resources (possibly due to context switch overhead), memcached cannot process the queries as fast as before and need to spend more time on computation, resulting in overall higher latency and saturation point appearing earlier.

(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?

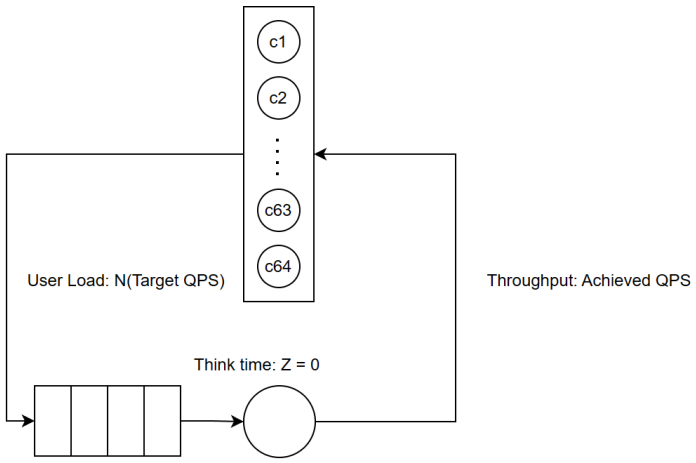
The `taskset` command sets CPU affinity of memcached and iBench benchmarks, to make them run on the same or different cores. The iBench benchmarks that run on the same core as memcached are cpu, l1d, l1i, l2. They need to run on the same core as memcached because CPU resource L1 cache(data and instruction), and L2 cache are per-core resources, thus the benchmarks have to run on the same core as memcached so it can use these resources of the specific core and introduce interference. The benchmarks that run on different cores are llc and membw. Since last level cache and memory are shared among all cores, running benchmarks on a different core as memcached can control the source of interference to be just from LLC or memory bandwidth, without introducing extraneous sources of interference, such as cpu, l1d, l1i and l2.

(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.

L1D, L2, LLC, and memory can be safely collocated with memcached. As indicated by the graph, memcached with these types of interference has 95th percentile latency below 1ms at 65K QPS. On the other hand, it is not safe to collocate CPU and L1I with memcached, as with such interferences memcached won't even be able to handle 65K QPS.

(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.

The experiment above is a closed system with 64 clients(4 connections per thread) as suggested by memcache-perf GitHub page. A diagram of the system is shown below:



The average response time can be calculated using the following expression:

$$R = \frac{N(QPS_{target})}{QPS_{achieved}}$$

The equation is derived from interactive law for closed system $R = \frac{N}{X} - Z$. In this case, think time Z is 0 as memache-perf's GitHub page suggests a request is sent immediately after receiving the response. N here should be defined as user load, which is the total number of queries/jobs submitted by all clients in one second. This value varies depending on the target QPS, thus we express it as a function of QPS_{target}. Finally, X is the number of queries the that the system is able to process per second, which is the achieved QPS.

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	1.32	1.21	1.65	1.24	1.48	1.42
canneal	1.00	1.10	1.13	1.27	1.12	1.67	1.20
dedup	1.00	1.68	1.27	2.08	1.30	2.11	1.71
ferret	1.00	1.91	1.07	2.31	1.01	2.51	1.97
freqmine	1.00	2.00	1.08	2.05	1.05	1.81	1.58
radix	1.00	1.04	1.03	1.12	1.02	1.57	1.02
vips	1.00	1.67	1.58	1.94	1.61	1.94	1.70

   The table shows blackscholes is most sensitive to L1I interference with a normalized execution time of 1.65, somewhat sensitive to CPU and memBW interference, and not very sensitive to L1D and L2 interference. The only resource interference canneal is sensitive to being LLC, while the other interference types result in normalized execution times well below the 1.3 mark, indicating insensitivity. Dedup is very sensitive to L1I and LLC interference, giving normalized execution time over 2. It is somewhat sensitive to CPU and memBW interference with a normalized execution time of around 1.7. Ferret is very sensitive to L1I and LLC interference with normalized execution times well over 2. It is also pretty sensitive to CPU and memBW interference as the normalized execution times are very close to 2. It is insensitive to L1D and L2 interference as the normalized execution times are both very close to 1 (no interference). Freqmine is sensitive to L1I interference with normalized execution time over 2, as well as CPU and LLC interference with close-to-2 normalized execution times. Similar to ferret, it is not sensitive to L1D and L2 interference, both have minimal effect on the normalized execution time. Radix is only sensitive to LLC interference, and has demonstrated no significant slowdown for all other interference types. Vips has noticeable performance degradation for all types of interference as the normalized execution times are all well above 1.3, but it is more sensitive to L1I and LLC interference, with normalized execution times close to 2.

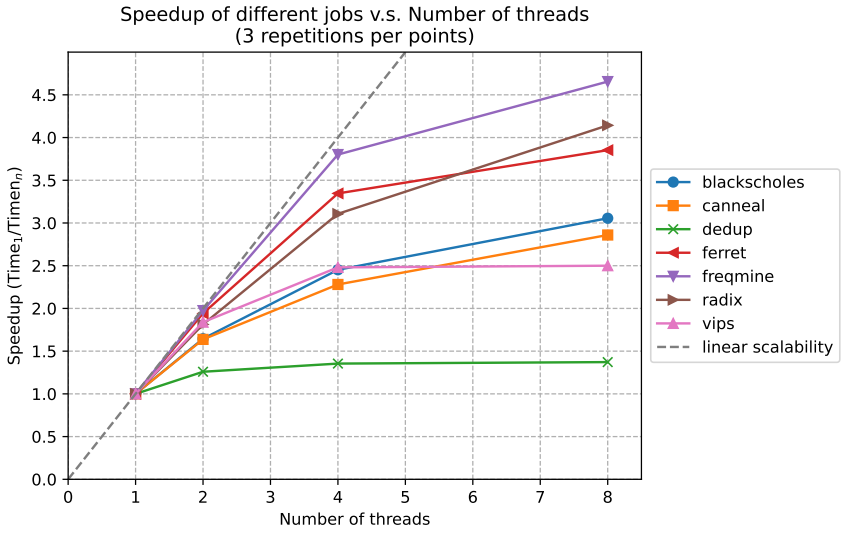
   (b) **[8 points]** Explain what the interference profile table tells you about the resource requirements for each application. Which jobs (if any) seem like good candidates to collocate with memcached from Part 1, without violating the SLO of 2 ms P95 latency at 40K QPS?

   The table indicates blackscholes has little requirements for L1D and L2 resources while having higher requirements for CPU, L1D, and memBW resources. Dedup and ferret require a lot of CPU, L1I, LLC, and memBW resources while having little requirements for L1D and L2 resources. Similarly, freqmine requires large amounts of CPU and L1I resources but have lower resource requirement for memBW, and has little L1D and

L2 resource requirements. Canneal and radix both have little requirements for CPU, L1D, L1I, memBW resources and have a bit higher requirements for LLC. Vips has intermediate requirements for all resources and higher requirements for L1I and LLC resources. The graph in Part 1 indicates to achieve SLO of 2ms P95 latency at 40K QPS, we cannot have interference on CPU and L1I resources, as with such interference the latency will be beyond 3ms at 40K QPS. Thus, we need jobs that require very little CPU and L1I resources to minimize the interference when collocated with memcached. This leads us to canneal and radix, as they both have little CPU and L1I requirements. They also have relatively low requirements for other resources, further reduce the impact of interference.

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".



From the graph, we can see that all applications have some speedup as the number of threads increases. However, all those speedups are under the dashed line which shows the linear scalability. Therefore, we can conclude that the scalability of all applications is sub-linear.

freqmine, **radix** and **ferret** gain significant speedup when the number of threads increases. Here we regard "significant speedup" as the slope of the speedup line larger than 3/4 when the number of threads increases from 1 to 4 and reaches a value larger than 3.5 with 8 threads. **blackscholes** and **canneal** also gains relatively big speedup but not significant. In addition, **vips** gains some speedup when increasing the number of threads from 1 to 4, but no extra speedup is gained with further increase and even a little dropdown. **dedup** is not much speed up when the number of threads increases.

We can see all the speedup increases when the number of threads increases from 1 to 2, and the speed of speedup becomes slower when more threads are involved. The bottleneck is probably the non-parallelizable sequential parts of the applications. In addition, increasing the number of threads also introduces overhead such as costs for resource allocation and thread coordination.