Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Cloud Computing Architecture

Semester project report

Group 21
Yiming Wang - 21-959-606
Dominic Steiner - 17-918-988
Adrian Jenny - 18-922-120

Systems Group
Department of Computer Science
ETH Zurich
April 11, 2022

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 [20 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 80000 in increments of 5000:

```
$ ./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 5000:80000:5000
```

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 80K). (note: the actual achieved QPS, not the target QPS)
- 95th percentile latency on the y-axis (the y-axis should range from 0 to 10 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:
See figure 1. The displayed measurements were averaged across three independent runs and the error bars show the variability of the results via the standard deviation.

(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.

Answer:
First, we note that for all measurements, the variability in the results started increasing only after each type of interference reached close to its saturation point. Before that point, the variability is practically zero. The measurements for interference on the **CPU** and the **L1 instruction cache** are strikingly similar. In both cases, for lighter workloads, the interference causes an increase in response time on the 95th percentile (tail latency) of up to about 4ms compared to no interference at all. Further, we see that the system starts to get saturated at about 35'000 QPS and that it cannot achieve more than 40'000 QPS. In that saturated state, the tail latency can be as much as 8ms or even slightly above. We further note that we get a much larger variation in response time across independent measurements as soon as the system is saturated. Memcached is a latency-sensitive application that makes heavy use of the instruction cache. Interference on the CPU or instruction cache itself will cause poisoning of the instruction cache and will therefore lead to a lot

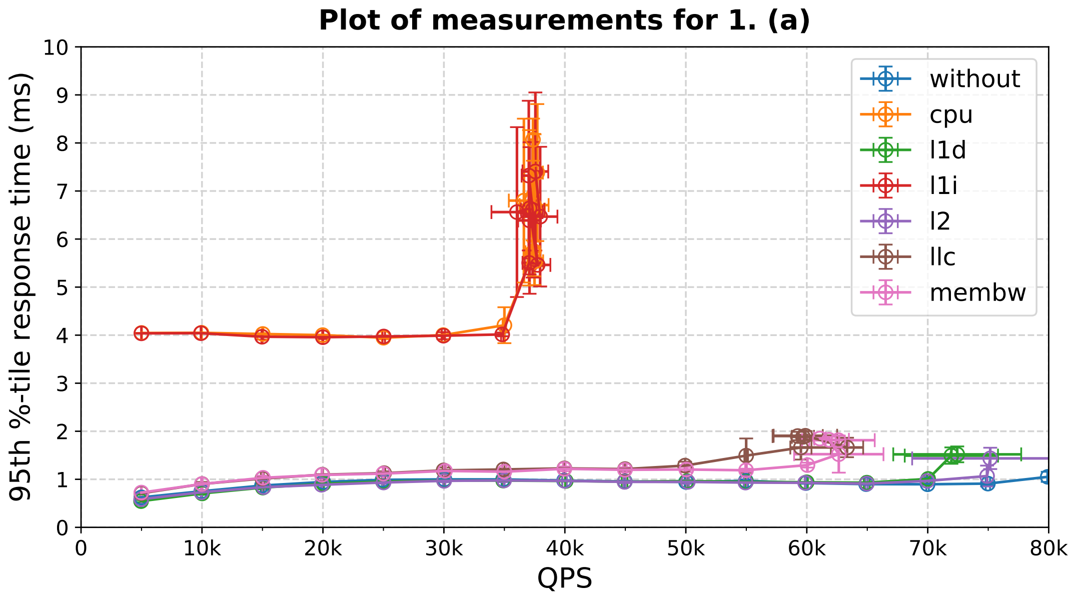

Figure 1: Plot of measurements for 1. (a)

of either cache conflict or capacity misses that need to be served by lower-level caches.
Interference on the **L1 Data Cache** (L1D), **L2 Cache**, **Last Level Cache** (llc) and **Memory Bandwidth** all yield relatively similar results compared to no interference (without), until each type of interference causes a saturation. At the saturation point, the achievable QPS become much less predictable and we have comparatively more variation in QPS than tail latency. This is exactly the opposite of what we have seen with CPU and L1i interference.

At the saturation point, llc and memory bandwidth interference increase the tail latency to about 1.8 ms. Under llc interference starts to reach its saturation point at about 55'000 QPS, closely followed by memory bandwidth interference at about 60'000 QPS. Interference on the l1d and l2 cache both yield systems that perform practically equal to a system with no interference up to 70'000 QPS. At this point, the system with interference on l1d reaches its saturation point closely followed by the system with l2 interference at around 75'000 QPS. At this stage, both of these systems' tail latency is just about 1.5ms however with considerable variance in achieved QPS. With interference on the data in the L1 data and L2 cache there are bound to be a lot of cache misses. It could be that memcached has to work with some global data structure for all queries that is accessed very often and would otherwise remain in cache if there were no interference. We also observe that for a system with no interference we had a slight increase in the tail latency at the beginning with it being mostly stable at 1ms. We further notice a slight increase just at the point where it reaches 80'000 QPS. This might be where this system is about to reach saturation as well.

In the experiment, all queries originated from a client on a different machine. In this case, all requests have to go through the complete network stack on the system. Memcached is designed as a key-value store for small chunks of data. This workload causes a lot of random memory reads and writes but the overall memory bandwidth required is quite low. This explains why memory bandwidth interference has a relatively low impact on latency. Such random access patterns exhibit low temporal and spatial locality, reducing the effectiveness of data caches. Together with the fact that we measured tail latency, which probably includes a few cache misses even without interference, explains that all types of data cache interference has a very low effect on latency. The llc has the biggest capacity of all the caches and is, therefore, the cache with the biggest effect on latency while still being quite small.

(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?

Answer:
The taskset command, in this case, is used to bind a specific command to a specific CPU core or set of CPU cores. We see that the interferences for the Last Level Cache (llc) and memory bandwidth are run on a different core than memcached whereas all other interferences are run on the same core as memcached. For the interferences that run on the same core, this is necessary since these sources of interference are local to one specific core. Obviously, CPU interference needs to run on the same core otherwise it would not affect memcached. The l1 and l2 caches are local to each core (on most modern machines at least) and poisoning these caches on another core will not lead to any interference for the core running memcached. As for why the benchmarks for llc and memory bandwidth are run on a different core, this is most likely done in order to rule out variables in the measurement process. If we were to run interference on the shared llc or memory bandwidth from the core that is also running memcached, then this would also poison the l1 and l2 caches as well as cause interference on the CPU core itself, thus not properly isolating the effects of these types of interference.

(d) [**2 points**] Assuming a service level objective (SLO) for memcached of up to 1.5 ms 95th percentile latency at 65'000 QPS, which iBench source of interference can safely be collocated with memcached without violating this SLO? Briefly explain your reasoning.

Answer:
According to Figure 1, the iBench source of interference **L1 Data Cache** (l1d) and **L2 Cache** (l2) can be collocated without violating the given SLO. It is because the l1d and l2 have around 1.0 ms P95 response time given the QPS of 65'000, not derivating a lot from the response time without any interference. Meanwhile, all other sources of interference reach their saturation points with QPS lower than 65k or need more than 1.5 ms, and thus fail to meet the SLO.
This outcome can be explained by the following facts.
The fact that interference on l1d and l2 does not affect the tail latency as much as other sources of interference suggests that the application cannot benefit as strongly from these fast caches. This is most likely due to their rather limited capacity and the fact that memcached deals with a comparatively large amount of data that will lead to lots of cache misses anyway.

Part 2 [25 points]

1. [**12 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. Summarize in a paragraph the resource interference sensitivity of each batch job.

   Answer:

Workload	`none`	`cpu`	`l1d`	`l1i`	`l2`	`llc`	`memBW`
dedup	1.00	1.50	1.25	2.21	1.25	2.17	1.67
blackscholes	1.00	1.42	1.40	1.91	1.38	1.73	1.42
ferret	1.00	1.93	1.11	2.79	1.06	2.74	2.13
freqmine	1.00	2.00	1.06	2.04	1.06	1.87	1.52
canneal	1.00	1.35	1.36	1.72	1.37	2.20	1.52
fft	1.00	1.32	1.30	1.97	1.31	2.16	1.55

   The dedup workload is most sensitive to l1i and llc interference causing more than 2 times slower execution times. CPU and memBW have a moderate, while l1d and l2 have a small effect on execution times.

   Blackscholes is still most affected by l1i and llc interference executing 1.91 and 1.73 times slower respectively. However, this is only slightly slower than all the other interference types which slow down the workload about 1.4 times.

   The ferret workload is most sensitive to l1i, llc, and memBW interference all of which cause the workload to take more than twice the time. Only slightly better handles ferret cpu interference, l1d and l2 interference cause barely any slowdown.

   Freqmine is again very sensitive to l1i interference, followed closely by cpu and llc. MemBW has moderated effect causing 1.52 times slower execution time. Again l1d and l2 have almost no effect.

   The canneal and fft workloads behaved quite similar to each other. Both are very sensitive to llc interference and moderately sensitive to l1i and memBW interference. All the other interference types only have a small effect on the execution times of there to workloads.

2. [**3 points**] Explain in a few sentences 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 1.5 ms P95 latency at 65K QPS?

   Answer:
   (a) When we look at how the execution times changed depending on the type of interference applied we get an estimate as to how much a given workload is affected by a specific source of interference. Generally, we can say that the longer the execution time gets compared to an execution with no interference, the more heavily a workload relies on having a significant part

of that resource for itself without interference from other programs. For example if we take the ferret workload we see that it was significantly affected by interference on the CPU, the instruction cache, the llc and the memory bandwidth. It would therefore be beneficial to have ferret run on a system where it does not need to share its CPU core with another application (to avoid interference on CPU and instruction cache) and at the same time the computations running on other cores should not make heavy use of the llc or memory bandwidth.

(b) When it comes to collocating a PARSEC workload with memcached we need to differentiate based on the meaning as to how we interpret “collocation”. In general we can say that we would only want to collocate workloads that do not interfere with each other too much. For example, if we had two applications that react very strongly to interference on the memory bandwidth then we would not wish to collocate them on the same system. When we talk about collocation in the sense that we let the PARSEC workload on a different CPU core than memcached, then we don't have to worry about interference on the CPU, l1d, l1i or l2. We would only need to worry about interference on the llc or memory bandwidth. We saw that for interference on the llc or the memory bandwidth that memcached's tail latency might still lie below 1.5ms at 65'000 QPS if we account for the shown margin of error. Because we think that the iBench sources of interference hit the resources rather hard we further note that we generally don't expect memcached or a given PARSEC workload to be quite as affected from each other's interference as was shown with the benchmark. Therefore, in the case of collocation on different CPU cores we would recommend to use the blackscholes workload for collocation since it is the least affected by interference on the llc and the memory bandwidth.

When we talk about collocation on the same CPU core with constant context switching we will also need to take interference patterns on CPU, l1d, l1i and l2 into account. We have seen in figure 1 that memcached can tolerate interference on the l1d and l2 cache relatively well whereas it does not work well at all with interference on the CPU or the instruction cache. We therefore wish to choose a PARSEC workload to colocate with memcached that does not make heavy use of the CPU or the instruction cache while we don't need to care as much about whether the PARSEC workload makes heavy use of the l1d or l2 cache. Unfortunately, none of the PARSEC workloads reacted particularly well to interference on the instruction cache with canneal being the least affected at only 1.72 times the execution time compared to no interference at all. Blackscholes provides the better overall balance but since we wish to primarily minimize the effects on the instruction cache and the CPU we would choose canneal as a potential candidate. This choice is also based on the above assumption that we don't expect the collocation of these workloads to hit the resources to the same degree as iBench.

3. [**10 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. Briefly discuss the scalability of each application: e.g., linear/sub-linear/super-linear. Do any of the applications gain a significant speedup with more threads? Explain what you consider to be “significant”.

Answer:
The dashed grey line in the Figure 2 indicates a fictional workload with perfect linear scaling. All the benchmarked workloads are below that line and therefore have sub-linear scaling. Still, most workloads scale pretty well to three threads. Canneal has the worst scaling but still achieves a speedup of two. We would say all workloads gain a significant speedup when executed with three threads. However, when working with six threads only freqmine, canneal

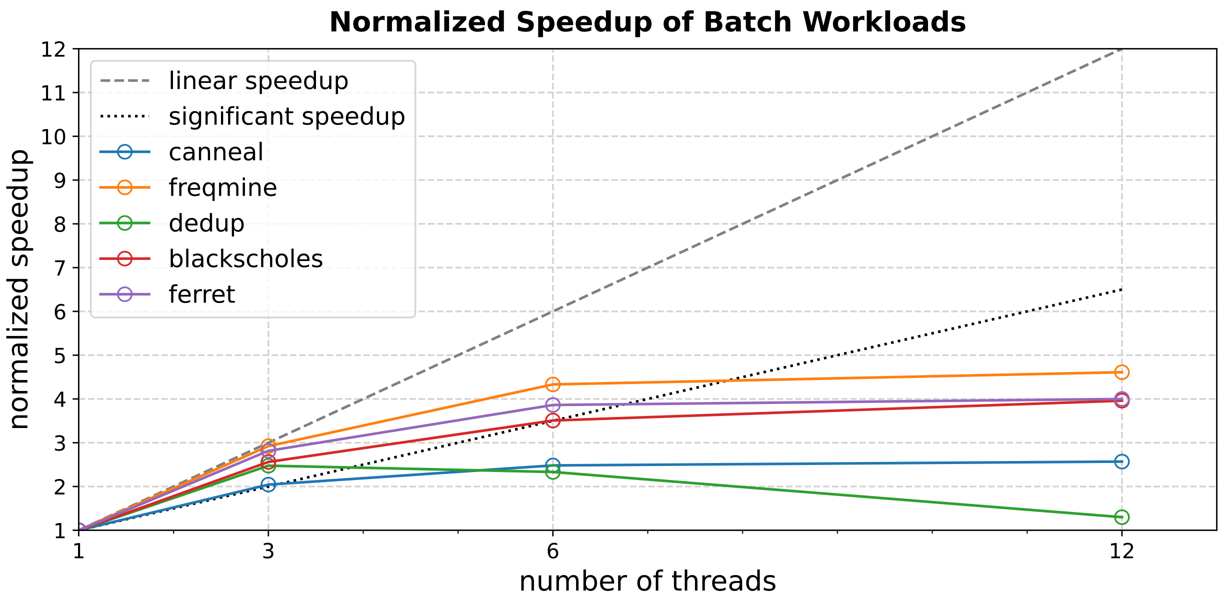

Figure 2: Plot of question 3. The fft workload only runs with thread numbers equal to powers of two, therefore we omitted fft from the plot.

and blackscholes can speed up significantly compared to three threads. Increasing the thread count further to 12 threads does not increase the performance significantly for any of these three workloads.

Looking at the remaining two workloads, canneal's performance does not increase significantly with more than three threads. Dedup's performance decreases while increasing the threads from three to six or 12 threads.

As to what constitutes a significant speedup, this is of course highly domain and application specific. In a domain where available resources are not an issue but every piece of additional speedup is very important then even a very low speedup may be significant despite the diminishing returns on investment. It is therefore difficult to to give a general statement.

Workloads that scale poorly waste more and more time on communication and synchronization overhead. We define a speedup as significant if the threads spend at least half their time doing useful work. This is the case if a workload achieves at least 50% of the idealized linear speedup. The dotted line in figure 2 shows this 50% border. All points above this line did achieve a significant speedup.