Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Systems@ETH zürich

Cloud computing architecture

Semester project report

Group 5
Berke Egeli - 20-948-220
Floris Westermann - 16-944-662
Julia Bazinska - 20-942-843

Systems Group
Department of Computer Science
ETH Zurich
June 16, 2021

Instructions

Please do not modify the template, except for putting your solutions, names and legi-NR. Parts 1 and 2 should be answered in maximum six pages (including the questions). **If you exceed the space, points might be subtracted**.

Part 1 [20 points]

(a) **[10 points]** Plot a single line graph with 95th percentile latency on the y-axis (the y-axis should range from 0 to 10 ms) and QPS on the x-axis (the x-axis should range from 0 to 55K). Label your axes. State how many runs you averaged across (we recommend three) and include error bars. There should be 7 lines in total in your plot, showing the performance of memcached running with no interference and six different sources of interference: cpu, l1d, l1i, l2, l3, membw. The readability of your plot will be part of your grade.

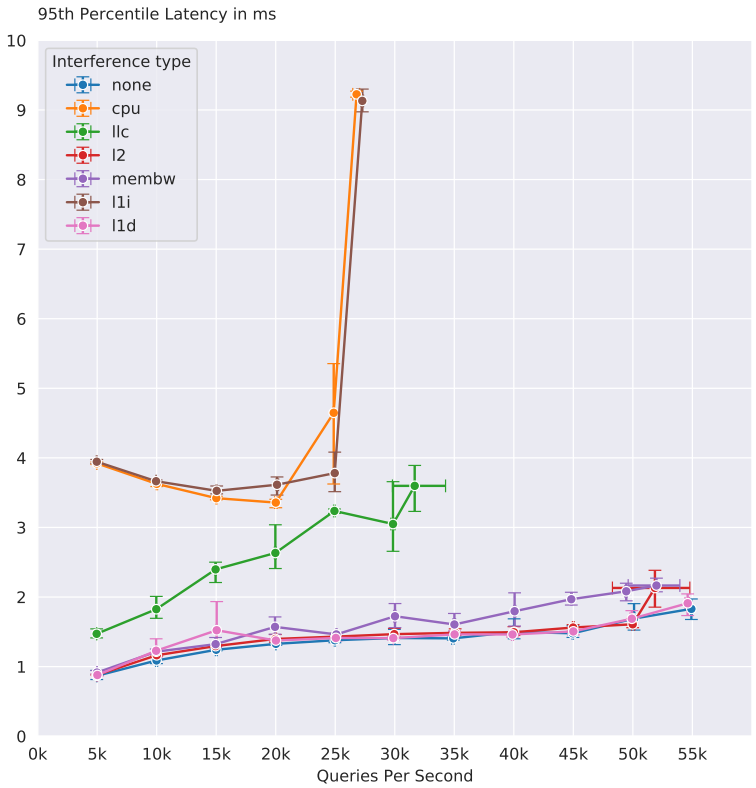

Figure 1: 95th percentile latency as a function of QPS handled by memcached, with the interference of iBench stressing different resources. Every datapoint is averaged across 3 runs. Whiskers represent 95% confidence intervals. For interferences `cpu, llc, l1i` the performance of memcached is so strongly impaired that the top target QPS values ($\geq 30k$ for `cpu, l1i`, $\geq 35k$ for `llc`) are not achieved.

(b) **[6 points]** Describe how the tail latency and saturation point (the "knee in the curve") of memcached is affected by each type of interference. Also describe your hypothesis for why memcached performance is affected in this way.

As can be seen in Figure 1, memcached is influenced the most by CPU and L1 instruction cache inteference. A significant drop in performance for both can be observed between 25k and 30k QPS. The service is not able to handle more than 28,000 QPS. This suggests that these types of interferences are the most critical and impact both latency and throughput.

Neither the L1 data or L2 cache interference benchmarks lead to a significant drop in performance. We assume it is caused by the fact that the same piece of data does not get accessed

by memcached often enough, which means it gets evicted from L1 and L2 caches between accesses even without interference. For the L2 interference we observe a drop in performance for a target kQPS of 55. The service saturates the throughput and is not able to provide 55 kQPS in a stable way. This might be because with high QPS some meta-information about requests etc. occupies more space and does not fit in L2 cache anymore.

On the other hand, the L3 cache is usually much bigger (at least one order of magnitude in Intel CPUs) and thus there is a chance the data residing there might be reused in queries. We think this is the reason why a drop in performance can be observed when the L3 cache interference benchmark is running. Because of the interference, the data gets evicted from the L3 cache and there are few or no cache hits. That slows down the process considerably, because memory accesses are much slower. We believe this causes the fact that the service is not able to handle the requested QPS – maximum throughput varies between 30 and 35 kQPS.

The last interference to consider is the memory bandwidth benchmark. Memcached, being a KV store, is a data oriented application and so it could be expected that using up memory bandwidth would make the latency much worse. However, the latency increase is not as big as for L3 congestion. As explained in the previous paragraph, the L3 cache is leveraged to omit memory accesses quite extensively. We suppose that memory bus congestion causes the accesses to be maybe a couple times slower. In comparison, a cache miss usually leads to a latency of one order of magnitude bigger than a cache hit.

(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 allows to set the CPU affinity of a process. For the benchmarks `cpu, l1d, l1i, l2` the interference process is set to the same core as memcached, while in `l3, memBw` it is bound to a different core. This is done because the L3 cache and memory bus are shared across cores. Thus executing on a different core still causes interference, while minimizing the impact on other resources. For the rest of the benchmarks, the contested resources are not shared between cores and thus the interference benchmarks need to run on the same core.

(d) **[2 point]** Assuming a service level objective (SLO) for memcached of up to 2 ms 95th percentile latency at 40K QPS, which iBench source of interference can safely be collocated with memcached without violating this SLO? Explain your reasoning.

According to our experimental results, collocating any of inference benchmarks for L1 data cache or L2 cache with memcached still provides 95th percentile latency of below 2ms. Memory bandwidth could also be considered, but it showed a higher variance in our experiments, with the confidence interval slightly exceeding 2ms. Because of this, memory bandwidth is not a safe choice.

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. 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.

Workload	none	cpu	l1d	l1i	l2	llc	memBW
dedup	1.0	1.57	1.37	2.23	1.28	2.26	1.73
blackscholes	1.0	1.44	1.49	1.89	1.35	1.77	1.47
ferret	1.0	2.06	1.07	2.79	1.08	2.91	2.21
freqmine	1.0	2.04	1.06	2.05	1.06	2.0	1.65
canneal	1.0	1.55	1.5	1.78	1.6	2.64	1.75
fft	1.0	1.46	1.39	1.95	1.43	2.21	1.66

All of the applications are heavily affected by interference on last-level cache. This is because when the L3 cache is congested, the applications need to transfer data from the main memory which is significantly slower than the caches. We do not see a similar pattern on the L1 and L2 data caches however. This is due to the fact that accessing the next level of cache is not orders of magnitude slower than accessing to memory from last-level cache in terms of cycles.

Ferret and Freqmine applications slow down the most when there is interference on CPU and L1 instruction cache, so it is safe to assume they are computation intensive applications. For both of the applications, performance does not change much when there is interference on L1 and L2 data caches but they both slow down when there is interference on L3 cache and memory bandwidth. This suggests that the working set of the applications are already big enough that they do not fit into L1 and L2 data cache but once they do not fit into L3 cache, or in the case of Ferret when there is congestion on memory bandwidth, they slow down considerably. Ferret is probably the most resource intensive application amongst all other batch jobs.

For all of the applications except Ferret when there is interference on memory bandwidth the execution time increases but the increase in execution time still falls in the orange bracket. This suggests that there is data locality in the applications and the caches help with data accesses. However, since there is still a non-negligible slowdown in applications, it means they still need to access data from the memory and the caches are not enough. Because of these memory accesses the performance still goes down.

Dedup seems to slow down most when there is contention in L1 instruction cache, L3 cache and memory bandwidth, so it is probably a memory intensive application. Blackscholes appears to slow down at similar rates for all of the interferences, so it does not favor CPU over memory or vice versa too much. Like all other batch jobs, it is slowest when there is interference in L1 instruction cache and L3 cache. Blackscholes is probably the least resource intensive batch job amongst all the others. Canneal suffers the most from interference in instruction cache and L3 cache as well. Next biggest slow downs are when there is interference in memory bandwidth and L2 cache, so it is most likely a memory intensive job. FFT is a low operational intensity application, so it is more likely to be bounded by memory than CPU. Again, like all batch jobs, it is slowest when there is interference on instruction and last-level caches. But it is next slowest when there is interference on memory bandwidth. For CPU and L1/L2 data caches, it has similar performance.

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 2 ms P95 latency at 40K QPS?

Dedup requires space in L1 instruction cache and last-level cache to operate efficiently. The performance drops when there is interference in CPU, L1 data cache, and memory bandwidth but the slow down still falls in the orange bracket, so interference in these resources is acceptable. It appears that Dedup is marginally affected by interference in the L2 cache as the slowdown falls in the green bracket.

Blackscholes executes acceptably well when there is interference in each of the resources. This suggests that it is not a very resource intensive application. However, it slows down the most when there is interference in the L1 instruction cache.

Ferret slows down considerably when there is interference in almost all of the resources. It is probably both computation and memory heavy. Interference in the L1 and L2 caches do not seem to affect the job significantly. This is suggests that application already has a lot of L1 and L2 cache misses when there isn't any interference, so the performance is not affected a lot when there is interference in those resources.

Freqmine seems to slow down considerably only when there is interference in the L1 instruction cache and CPU, so it is probably a computation heavy job.

Canneal and FFT seem to have similar resource requirements and they both slow down when there is interference in LLC.

From the first part we know that memcached performance suffers considerably when there is interference in CPU, L1 instruction cache and LLC, so, any application that is heavy in those resources is not safe to collocate with memcached. It is acceptable if the applications predominantly require L1 data cache, L2 cache and memory bandwidth. Blackscholes is perhaps the safest batch job to collocate with memcached. It is not CPU intensive and does not congest memory bandwidth considerably. However, it still suffers from L1 instruction cache misses like all other jobs. Despite this, since it falls in the orange bracket we consider its use of L1 instruction cache to be moderate. Both Canneal and FFT would heavily congest LLC and because memcached also uses LLC heavily, it is not safe to collocate these jobs with memcached. Ferret uses CPU, L1 instruction cache and memory bandwidth intensively, so it cannot be collocated with memcached. Freqmine is similarly CPU and L1 instruction cache intensive. Dedup is not a good option either, because it heavily uses L1 instruction cache.

3. **[10 points]** In a Kubernetes cluster with a single 8-core node, run each of the six batch jobs individually and measure their execution time. Note that you should not use any interference for this part of the study. Vary the number of threads (1, 2, 4, and 8). Plot a single line graph with the speedup on the y-axis (normalized time to the single thread performance: $\frac{t_1}{t_n}$ where t_i is the execution time for i threads) and the number of threads on the x-axis. Briefly discuss the scalability of each application, mentioning if it is linear, sub-linear or super-linear. Which of the applications, if any, gain a significant speedup with more threads? Explain what you consider to be "significant". The setup for this question is detailed in Section 2.1.2.

   The results of our measurements can be seen in Figure 2. `freqmine` benefits most from parallelization. The speedup scales nearly linear up to 4 threads. Increasing the threads from 4 to 8 still yields an improvement. However, at less than a 25% increase it is significantly smaller than before.

   A similar trend is noticeable for all other applications. `ferret` and `backscholes` appear to scale linearly up to 2 threads. After that they both scale slightly sub-linearly, with again a noticeable flattening of the curve for 8 threads.

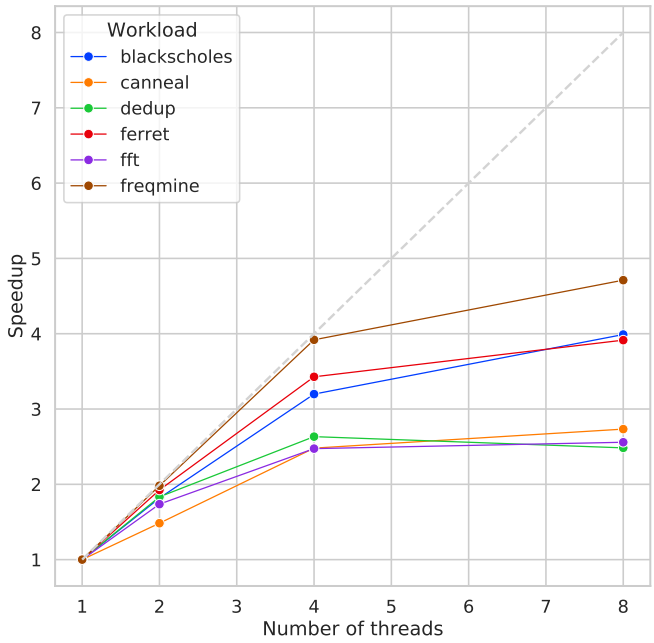

Figure 2: Speedup of specified workloads on a single 8-core node.

`fft` and `dedup` scale close to linearly for 2 threads but scale significantly worse than `ferret` and `backscholes` after that. `dedup` loses performance when run with 8 threads compared to 4 threads. `canneal` scales worse than the previous two for 2 threads, but behaves similarly to `fft` beyond that point.

Speedup through parallelization is problem dependent and not all applications can reach linear speedup, independent of the implementation. However, we decide to categorize the measured speedups as follows:

- Linear speedup is ideal (as would super-linear speedup if we had encountered it).
- A speedup of more than $\frac{3}{4} * \#Threads$ is good.
- A speedup of more than $\frac{1}{2} * \#Threads$ is still significant.
- Anything less is insignificant.

Thus, `freqmine`, `blackscholes`, and `ferret` have significant scaling up to 8 threads. The others have significant scaling only up to 4 threads.