Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Cloud Computing Architecture

Semester project report

Group 018
Aristotelis Koutris - 23942683
Georgios Manos - 23948342
Maria Tsanta - 22952857

Systems Group
Department of Computer Science
ETH Zurich

March 27, 2024

Instructions

- **Please do not modify the template!** Except for adding your solutions in the labeled places, and inputting your group number, names and legi-NR on the title page.

  Divergence from the template can lead to subtraction of points.

- Parts 1 and 2 of the project should be answered in **maximum 6 pages** (including the questions, and excluding the title page and this page, containing the instructions - the maximum total number of pages is 8).

  If you exceed the allowed 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 5000 to 55000 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 5000:55000:5000
```

Repeat the run for each of the 7 configurations (without interference, and the 6 interference types) **at least 3 times** (3 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 to make sure your cluster is deleted.

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

- Queries per second (QPS) on the x-axis (the x-axis should range from 0 to 55K). (**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.

Answer: 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? What is your reasoning for these effects? Briefly describe your hypothesis.

Answer:

- `ibench-cpu`:

  Tail latency(95th percentile latency) increases for smaller number of queries (up to 25K QPS) reaching up to 4ms versus the 1ms we get with no interference. After 25K QPS the system starts to get saturated and it cannot exceed 30K QPS. While saturated, the tail latency reaches 8ms or more.

  Explanation: Deploys a workload that issues an increasing number of integer operations [3] [5] that keep the integer processing units of the core busy. Increasing the number of queries on the cluster, leads to increased CPU workload. Thus, we observe increased latency on all the queries and reach saturation for smaller QPS.

- `ibench-l1d`: Up to 47K QPS the tail latency is reaching up to 1ms which is the same latency that we get with no interference. At about 47K QPS the system reaches its saturation point getting a tail latency of about 2.2ms while having increased variance in the QPS.

  Explanation: Deploys a kernel that sweeps through increasing fractions of the L1d until it populates its full capacity. Accesses are random [3] resulting in low temporal and space locality in cache accesses. Thus, we expect to see low impact on the latency across all queries.

- `ibench-l1i`: Similarly to the run with CPU interference, for up to 25K QPS there is an increase in tail latency reaching up to 4ms compared to the 1ms we get with no interference. For rates beyond 25K QPS the system starts to get saturated and it cannot exceed 30K QPS. While saturated, the tail latency reaches 8ms and can at times surpass that.

  Explanation: Deploys a kernel that sweeps through increasing fractions of L1i until it populates its full capacity. Accesses are random [3]. Increasing QPS leads to increased CPU workload. It is reasonable to have increased latency on all the queries as the instruction read latency will increase, but also reach the saturation point for smaller number of QPS, similarly to the CPU SoI.

- `ibench-l2`: The measurements for interference with L2 show us that for up to 49K QPS the tail latency is reaching up to 1ms which is the same latency that we get with no interference. At about 49K QPS the system reaches its saturation point getting a tail latency of about 2.2ms while having a bit of variance in the QPS.

  Explanation: Deploys a kernel that sweeps through increasing fractions of the L2 until it populates its full capacity. Accesses in this case are random [3], like `l1d`, this results in low temporal and space locality, reducing the effectiveness of all data caches. Thus, it is expected that we see low impact on the latency on across all queries.

- `ibench-llc`: The behavior is similar to the no interference runs up to 30K QPS, seeing as the tail latency is reaching up to 1.6ms. After 30K QPS the latency starts to increase, reaching the saturation point at 41K QPS and getting a tail latency of about 3ms. At the saturation point we can see some variance in the QPS in between runs.

  Explanation: Deploys a kernel that sweeps through increasing fractions of the LLC until it populates its full capacity. Therefore, like the `l1d` and `l2` SoIs, it is expected that we see low impact on the latency across all queries. Since LLC has the biggest capacity from all of the caches, there is a higher probability for a cache hit in random accesses, thus, regarding cache interference SoIs, LLC interference has the higher impact on latency but also the smallest QPS saturation point.

- `ibench-membw`: Like the LLC SoI, tail latency is resembling the no interference runs, up to 40K QPS, seeing as the tail latency is reaching 1.6ms. After 40K QPS the latency starts to increase, reaching the saturation point at 44K QPS and getting a tail latency of about 2.7ms. At saturation point we can see some variance in the QPS in between runs.

  Explanation: Performs streaming (serial) memory accesses of increasing intensity to a small fraction of the address space, until the SoI consumes 100% of the sustained memory bandwidth of the machine [3]. This is used to quantify the sensitivity of the workload to memory interference but the query results transfer occurs through the network (memcached server to client-agent). Thus, the workload is still sensitive to the memory bandwidth SoI, but the main latency cause is the network. This SoI doesn't have as big an impact as `llc`, since by not polluting the LLC, cache misses are kept low and we avoid having to access the memory.

(c) [**2 points**]

- Explain the use of the `taskset` command in the container commands for memcached and iBench in the provided scripts.

  Answer: `taskset` allows us to set the CPU core and number of threads that a process will use. The -c flag assigns a specific CPU core to the process, -t flag determines the number of threads it will use and -u flag sets the user under which the process is executed.

- Why do we run some of the iBench benchmarks on the same core as memcached and others on a different core? Give an explanation for each iBench benchmark.

  Answer: By assigning memcache and some of the SoIs to the same CPU core we can study interference patterns that rely on core specific bottlenecks. The `cpu`, `l1d`, `l1i` and `l2` SoIs, all utilize a single core's resources, so are assigned to the same core as the memcache process. Inversely, `llc` and `membw` SoIs, utilize the shared L3 and memory bandwidth respectively. By assigning them to a different core (core 1 here), we can isolate and measure the impact that the SoIs have due to the contested resources only.

(d) [**2 points**] Assuming a service level objective (SLO) for memcached of up to 2 ms 95th percentile latency at 35K QPS, which iBench source of interference can safely be collocated with memcached without violating this SLO? Briefly explain your reasoning.

On figure 1, we can see that L1d, L2, LLC and memory bandwidth SoIs can be safely collocated with memcached without violating the SLO. For 35K QPS, L1d and L2 SoIs result in a latency of 1ms, memory bandwidth SoI in a latency of 1.5ms and LLC SoI in a latency of 1.7ms. Meanwhile, with L1i and CPU SoIs, memcached workload has already reached its saturation point by 35K QPS and need more than 2ms (up 9ms) violating the SLO.

Answer:

(e) [**5 points**] In the lectures you have seen queueing theory.

- Is the project experiment above an open system or a closed system? Explain why.

  Answer: Our system uses `mcperf` to deploy agents that send requests to the `memcached` server synchronously, waiting for response before sending the next request. Therefore, the part of the system that involves the client-agent and the `memcached` server is a closed system. The `-D` flag in client-measures allows it to operate as an "open-loop" client [4].

- What is the number of clients in the system? How did you find this number?

  Answer: $N = N_A * N_T * N_C = 1 * 4 * 16 = 64$ clients, where `N_A` number of agents, `N_T` number of threads per agent and `N_C` is the number of connections per thread.

- Sketch a diagram of the queueing system modeling the experiment above. Give a brief explanation of the sketch.

  Answer: We have 64 clients (connections) to 1 memcached server. See Figure 2.

- Provide an expression for the average response time of this system. Explain each term in this expression and match it to the parameters of the project experiment.

  Answer: In closed system, response time is $R = \frac{N}{X} - Z$. R is the average measured latency in a steady state ($\lambda = \mu$), N is the number of clients, X is the QPS in steady state and $Z = 0$ thinking time. In the no interference measurements, applying the above formula in its steady state $R_{theory} = \frac{64}{50290.4} - 0 \approx 1.2726\ ms$ and $R_{measured} = 1.3320\ ms$, thus the results are really close.

Part 2 [31 points]

1. **Interference behavior [20 points]**

   (a) **[10.5 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 cell 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 less or equal to 2, and **red** if the normalized execution time is greater than 2. The coloring of the first three cells is given as an example of how to use the cell coloring command. **Do not change the structure of the table. Only input the values inside the cells and color the cells properly.**

Workload	`none`	`cpu`	`l1d`	`l1i`	`l2`	`llc`	`MemBw`
blackscholes	1.00	1.28	1.26	1.54	1.27	1.48	1.37
canneal	1.00	1.18	1.15	1.31	1.15	1.83	1.24
dedup	1.00	1.34	0.96	1.59	1.01	1.85	1.29
ferret	1.00	1.97	1.26	2.40	1.29	2.91	2.10
freqmine	1.00	1.96	1.55	2.07	1.03	1.87	1.55
radix	1.00	1.03	1.00	1.08	1.02	1.68	1.01
vips	1.00	1.72	1.67	2.03	1.65	2.41	1.68

   (b) **[7 points]** Explain what the interference profile table tells you about the resource requirements for each job. Give your reasoning behind these resource requirements based on the functionality of each job.

   Answer:

   - `blackscholes:` Moderately sensitive to L1i, LLC and memory bandwidth (MemBw). Less sensitive to CPU, L1d and L2.
     Explanation: Uses the simlarge set with 65k input options [1], thus the input is small enough for `LLC` but bigger than L1d and L2. It is limited by the amount of flops the processor can perform [1], thus resulting in low CPU utilization.
   - `canneal:` Moderately sensitive to LLC and L1i. Less sensitive to CPU, L1d, L2 and MemBw.
     Explanation: Makes use of cache-aware simulated annealing to reduce cache capacity misses. Thus, reduced sensitivity to MemBw and increased sensitivity to LLC is observed - as the input is still large enough for L1d and L2.
   - `dedup:` Moderately sensitive to LLC, L1i, CPU. Less sensitive to L1d, L2, MemBw.
     Explanation: Depends on a high number of read and write operations as it performs high compression ratio [1]. Due to large input size and CPU utilization the workload is insensitive to L1d and L2 SoIs, but sensitive to LLC and CPU SoIs.
   - `ferret:` Severely sensitive to LLC, L1i and MemBw. Moderately sensitive to CPU. Less sensitive to L1d and L2.
     Explanation: Performs content-based similarity search in a db of 35K images. Requires frequent main memory accesses and many flops. Workload is severely sensitive to LLC, L1i and MemBw SoIs. Also highly sensitive to CPU SoI.
   - `freqmine:` Severely sensitive to L1i. Moderately sensitive to CPU, LLC and L1d and MemBw workloads. Less sensitive regarding L2.

Explanation: Parts of the pipeline exhibits high temporal and spatial locality having many L1d hits while other parts require frequent LLC and memory accesses. Also there are many operations, leading to increased sensitivity to CPU and L1i SoIs.

- `radix`: Moderately sensitive to LLC workload, less sensitive to other resources.
  Explanation: Program is bound by integer operations in the CPU [2] and requires frequent writes without temporal locality. Thus moderate sensitivity to LLC and low sensitivity to other SoIs.
- `vips`: Severely sensitive to the LLC and L1i Cache, moderately sensitive to CPU, MemBw, L1d and L2 workloads.
  Explanation: Requires several linear operations in images, resulting in a high number of flops [1] and main and cache memory accesses. Thus, workload is severely sensitive to LLC and L1i SoIs. Also moderately sensitive to the rest SoIs.

(c) **[2.5 points]** 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? Explain why.
Answer: The memcached job is highly sensitive to CPU and L1i workloads. Therefore, `radix` would be a good candidate as it is not heavy on those resources. `blackscholes` and `canneal` although not heavy on the CPU, they are moderately heavy on the L1i.

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

(a) **[7.5 points]** Plot a 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). Pay attention to the readability of your graph, it will be a part of your grade.
Answer: See Figure 3

(b) **[3.5 points]** Briefly discuss the scalability of each job: 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".
Answer: Ideally, using t threads has a speedup of t (Amdahl's law). Not possible in practice as a program usually is not 100% parallelizable and because of the communication among threads overhead. We consider a speedup to be significant if for up to $t = 4$ we get 75% of the ideal speedup and for $t = 8$ above 50% of the ideal speedup.

- `blackscholes`: scales sub-linearly. Performance increases slightly for increasing t, up to a factor of x3.38 for 8 threads.
- `canneal`: scales sub-linearly. Performance improves slightly for increasing t, up to a factor of x3.08 for 8 threads.
- `dedup`: scales sub-linearly. Performance improves for up to $t = 4$ but drops for $t = 8$ as the overhead of splitting across threads is larger than the benefit.
- `ferret`: scales sub-linearly. Performance improves slightly for increasing t, up to a factor of x3.97 for 8 threads.
- `freqmine`: scales sub-linearly. Performance improves for increasing t significantly, up to a factor of x4.78 for 8 threads.
- `radix`: scales linearly up to 4 threads and then sub-linearly. Performance improves for increasing t significantly, up to a factor of x6.02 for $t = 8$.
- `vips`: scales sub-linearly. Performance improves for increasing t significantly, up to a factor of x4.43 for $t = 8$.

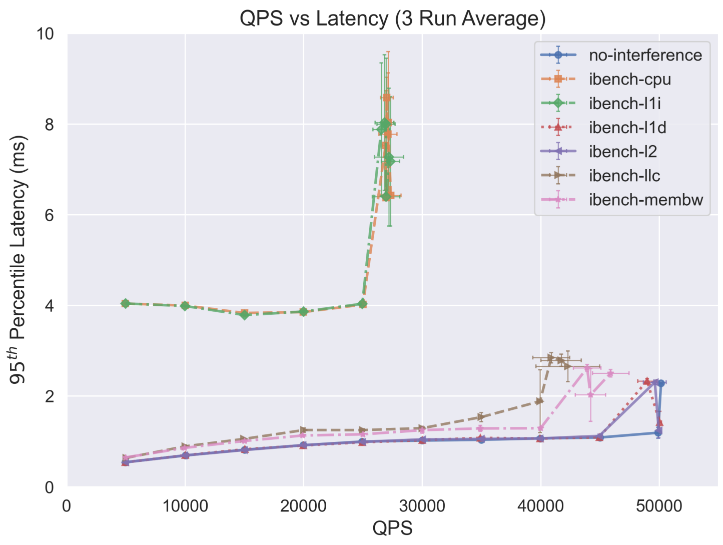

Figure 1: The y-axis upper bound was increased to 10 ms in order to include all points within the figure's bounds.

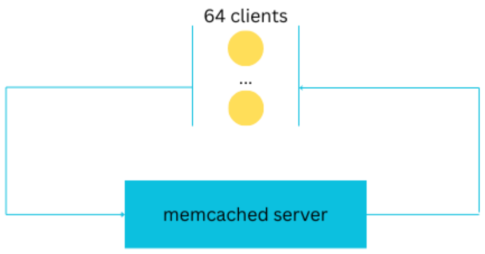

Figure 2: Diagram modeling our experiment.

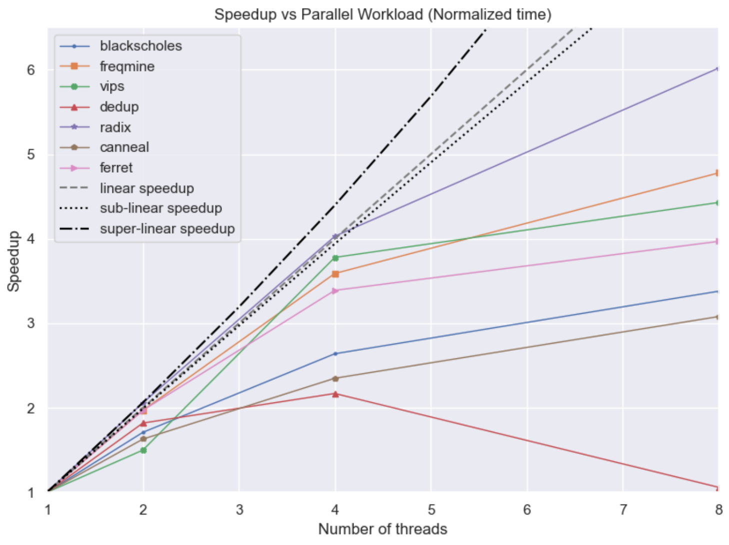

Figure 3: Speedup vs Number of threads t.

References

[1] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, page 72–81, New York, NY, USA, 2008. Association for Computing Machinery.

[2] Steven Cameron Woo, Moriyoshi Oharat, Evan Torriet, Jaswinder Pal Singh, and Anoop Gupta. The splash-2 programs: Characterization and methodological considerations. In *22nd annual international symposium on Computer architecture (ISCA '95)*, pages 24–36, 1995.

[3] Christina Delimitrou and Christos Kozyrakis. ibench: Quantifying interference for datacenter applications. In *2013 IEEE International Symposium on Workload Characterization (IISWC)*, pages 23–33, 2013.

[4] Shay Gal-On. Memcached Github Repository. `https://github.com/shaygalon/memcache-perf`.

[5] stanford mast. iBench Github Repository. `https://github.com/stanford-mast/iBench/tree/master`.