Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Cloud Computing Architecture

Semester project report

Group 076
Dennis Jüni - 19-935-493
Eric Nothum - 19-946-136
Alex Thillen - 19-946-953

Systems Group
Department of Computer Science
ETH Zurich
April 16, 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.

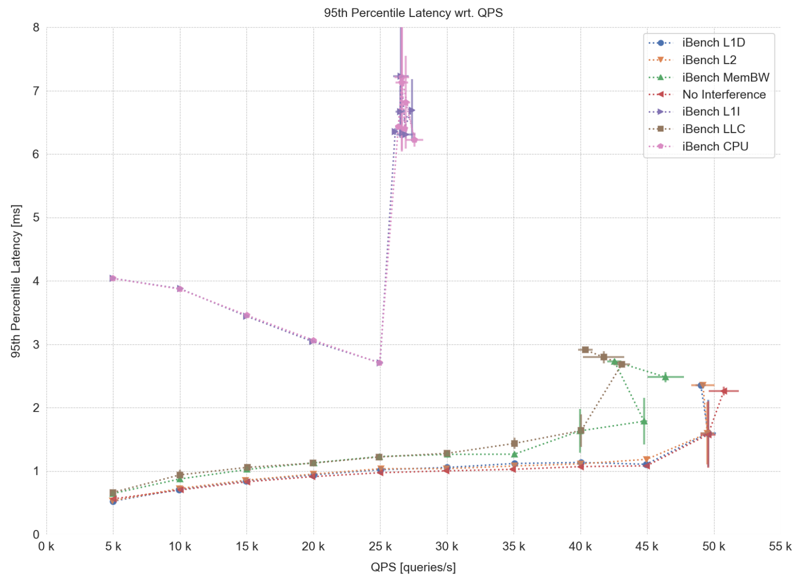

Figure 1: Latency of the 95-th percentile plotted against actual queries per second (QPS) averaged across 3 runs.

(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`: Saturation point reached at ca. $25k$ QPS. Tail latency increases drastically across all load levels, especially at levels above $25k$ QPS. E.g., at $10k$ QPS, the 99-th percentile latency is at ca. 5000ms vs. ca. 800ms without interference.

  Explanation: Memcached heavily relies on the CPU to process requests. CPU interference thus leads to a drastic degradation in performance.

- `ibench-l1d`: Saturation point reached between $45k$ and $50k$ QPS, similar to no interference. Tail latency progressively increases as load increases, but almost identically to no interference.

  Explanation: Memcached does not significantly benefit by the L1d cache. This might be because accesses may not exhibit good locality and the penalty for a L1d cache miss is small.

- `ibench-l1i`: Saturation point reached at ca. $25k$ QPS. Tail latency drastically increases across all load levels, again very similar to CPU interference.

  Explanation: L1i interference leads to cache misses when fetching the necessary instructions. This leads to degradation due to working with few, heavily-reused instructions.

- `ibench-l2`: Saturation point reached somewhere between $45k$ and $50k$ QPS. Tail latency progressively increases as load increases, but almost identically to no interference.

  Explanation: Similar to L1d cache, the accesses might still not exhibit strong locality in the relatively small L2 cache and the latency penalty for an L2 cache miss is still not significant.

- `ibench-llc`: Saturation point reached at ca. $40k$ QPS. Tail latency progressively increases as load increases, but is generally much larger than no interference (ca. $2-4$ times larger).

  Explanation: LLC is the last cache before (the much slower) memory. LLC interference will lead to many more cache misses in the LLC and thus to more (much slower) memory accesses.

- `ibench-membw`: Saturation point reached at roughly $40k$ QPS. Tail latency progressively increases as load increases, in almost the same way as LLC interference.

  Explanation: Suggests that memcached is constrained by memory bandwidth. When bandwidth is saturated by an interference, the memcached read and write operations are delayed.

(c) [**2 points**]

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

  Answer: `taskset` is used to set or retrieve a process's CPU affinity.[1] This allows us to choose if benchmarks are run on the same or on a different core as memcached.

- 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: Memcached runs on AMD EPYC CPU, which has an L1d and L2 cache per core, while LLC and memory is shared between all cores. To ensure the observed results

[1]Linux manual page

are caused by interference on LLC/MemBW and not due to running a job on the same core, we run LLC and MemBW interference on a different core. We run CPU, L1i, L1d, and L2 interference on the same core, otherwise it would not impact memcached at all.

(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.

Answer: L1d, L2, LLC, and MemBW interference can safely be collocated with memcached without violating the SLO. Running the above-mentioned interferences leads to a 95th percentile latency of less than 2ms at 35K QPS, even when including error or checking runs individually. There could potentially still be scenarios in which the latency increases above 2ms due to other unforeseen circumstances, however, for the given SLO this should be fine.

(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: The experiment is a closed system. In our mcperf setup we have one closed system agent (in `client-agent`), meaning that it waits for the requests to finish. The master node in the mcperf setup (in `client-measure`) is run with the flag `-D 4`. This allows the master node to operate as an "open-loop" client, while the other agents continue as "closed-loop" clients.[2] Specifically, it means that the master node can have up to 4 outstanding requests, instead of only one. Even though the master node does not fulfill the exact specifications of a standard agent in a closed system, all 4 primary criteria for a closed system introduced in the lecture are still fulfilled: Load comes from a limited set of clients; clients wait for response before sending next request; load is self-adjusting; system tends to stability

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

  Answer: The flags we use to setup the `mcperf` master and agent reveal the following:

  - `-T 16` : set the number of threads to 16 (both in the agent and in the master)
  - `-C 4` : set the master client connections per server and thread (overwriting `-c`)
  - `-c 4` : set the agent client connections per (`memcached`) server and thread.

  This gives a total of $(16 \cdot 4) + (16 \cdot 4) = 128$ clients or connections. This is also the result we obtained, when observing open connections to memcached using `telnet` and `stats`.

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

  Answer:

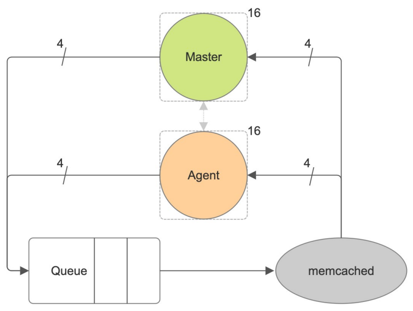

Figure 2: Sketch of the queuing system

[2]Memcache-Perf Github

The master and the agent use 16 threads, each with 4 connections. The master can have 4 outstanding requests per connection. The queue is not implemented directly within `memcached`, but rather in the network interface. The arrow between master and agent represents coordination taking place between them.

- 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:

  The Interactive Response Time Law states: $R = \frac{N}{X} - Z$

  In our case X is the system throughput. $N = 128$ is the number of clients. Z is the think time per client. We assume our load generator distributes the load evenly over a given time period. This allows us to calculate Z explicitly. If the service rate μ is lower than Q_{target}, the clients will immediately submit the next query, thus $Z = 0$. If the service rate μ is higher than Q_{target}, each client will have an average think time of $N \cdot (\frac{1}{Q_{target}} - \frac{1}{\mu})$. Combining the two cases, we end up with the following formula for Z:

$$Z = \max\left(0, N \cdot \left(\frac{1}{Q_{target}} - \frac{1}{\mu}\right)\right)$$

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	1.27	1.35	1.54	1.55	1.57	1.35
canneal	1.00	1.73	1.70	1.93	1.80	2.54	1.69
dedup	1.00	1.62	1.26	2.15	1.26	2.29	1.61
ferret	1.00	1.84	1.08	2.34	1.18	2.49	1.94
freqmine	1.00	2.00	1.03	2.00	1.04	1.92	1.59
radix	1.00	1.04	1.02	1.10	1.06	1.90	1.05
vips	1.00	1.59	1.58	1.78	1.55	1.91	1.62

   (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:` All interference types affect the execution time roughly equally (L1i, L2, and LLC having a slightly higher performance decrease). This suggests that all resources are roughly equally utilized and there is no single bottleneck.

Explanation: Blackscholes is a computationally-intensive, cache-sensitive benchmark using a small working set, which matches the balanced interference profile. L1d interference leads to more cache misses, but only small latency penalty. MemBW is not a bottleneck, due to the small working set fitting into the caches.[3]

- `canneal:` All interferences affect the execution time roughly equally, except for LLC which affects it heavily. This suggests that Canneal relies on the LLC significantly.
  Explanation: Canneal is specifically designed to have a highly-demanding memory behavior using a huge working set. Thus, LLC interference leads to more cache misses with large latency penalties. The demanding memory behavior also explains why other cache interferences and MemBW degrade the performance. The computation is still complex, thus reliance on the CPU is also explainable.[3]
- `dedup:` L1i and LLC interference intensively affect performance, while L1d and L2 do not affect it much. MemBW and CPU interference lead to slight performance degradation. Thus, Dedup relies heavily on the L1i and LLC, while L1d and L2 are not utilized much. The task also has a great CPU and MemBW utilization.
  Explanation: Dedup deduplicates and compresses data, which suggests that it reuses certain instructions often and thus utilizes the L1i a lot. Dedup works with a huge working set which explains the heavy reliance on the LLC. The job is still computationally intensive, explaining the performance degradation with CPU interference.[3]
- `ferret` The resource requirements are very similar to those of Dedup. It has slightly higher L1i, LLC, MemBW, and CPU usage, and slightly lower L1d and L2 usage.
  Explanation: Ferret is a content-based similarity search with a huge working set. Since the working set leads to many misses in small caches, their benefit is not significant CPU and MemBW reliance is larger than Dedup, since similarity search is more computationally intensive and Ferret has a slightly larger working set.[3]
- `freqmine:` Resource requirements are similar to Dedup, but LLC is less utilized and CPU & MemBW more. L1d & L2 interference have almost no impact on execution.
  Explanation: Freqmine searches patterns in a huge working set, which is computationally more intensive than Dedup, explaining CPU degradation. LLC interference has smaller impact, since cache misses are inevitable in the huge working set. L1d & L2 caches are not helpful at all since we miss frequently in these caches anyways.[3]
- `radix:` Only LLC interference affects performance much. The job is thus not bound by any of the resources (especially not by CPU and MemBW).
  Explanation: The working set of Radix is relatively small compared to other jobs, with straight-forward computation, explaining little CPU reliance. Radix sort processes data in a linear fashion, which likely mitigates the impact of interference with smaller caches. Interference in the LLC leads to many cache misses which incur a significant latency penalty, which explains the worse performance.[4]
- `vips:` All interferences affect the execution times of Vips roughly equally (L1i and LLC affect it slightly more). Thus, there is no single bottleneck within the resources.
  Explanation: Vips performs image transformations on a medium-sized working set. Since image transformations are complex computations, this explains the performance degradation with CPU interference. Since we have a medium-sized working set, all cache interferences roughly degrade the performance equally (LLC slightly more due to a higher latency penalty).[3]

[3]PARSEC Benchmark descriptions
[4]Radix Details

(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: Memcached did not satisfy the SLO under CPU and L1i interference, thus we do not want to collocate a job with high CPU or L1i cache utilization. Radix satisfies these requirements and would thus be the best candidate. Blackscholes could work as well, however, we would need to further test the exact implications of running it simultaneously.

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:

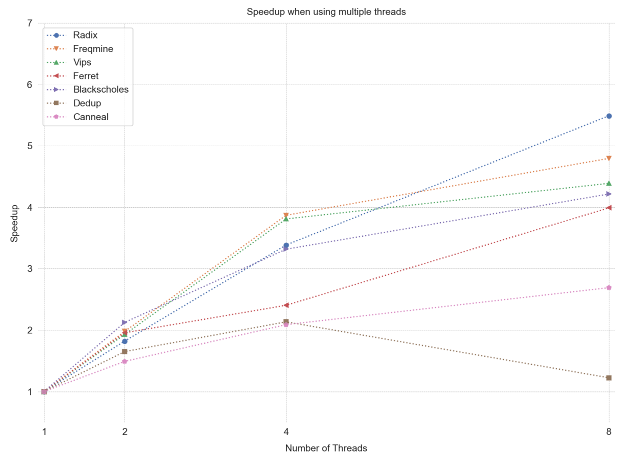

Figure 3: Speedup when using 1, 2, 4, and 8 threads on the 7 different workloads mentioned above.

(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: We define a significant speedup to be a speedup that exceeds 4 using 8 threads.

- `blackscholes`: This job scales sublinearly, even though going from 1 to 2 threads results in a superlinear speedup. The speedup is significant.
- `canneal`: This job scales sublinearly with no significant speedup.
- `dedup`: This job scales sublinearly with a very poor speedup overall.
- `ferret`: This job scales sublinearly from 2 to 4 threads. However, the additional performance per thread increases from 4 to 8 threads. The speedup is significant.
- `freqmine`: This job scales linearly for up to 4 threads, then diminishing returns become visible. Still the speedup of almost 5 at 8 threads is significant.
- `radix`: This job scales almost linearly going from 1 to 8 threads. The total speedup is almost 6.
- `vips`: This job scales linearly for up to 4 threads, then diminishing returns become visible. Still, the speedup is significant.