Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Systems@ETH Zürich

Cloud Computing Architecture

Semester project report

Group 020
Ambroise Aigueperse - 21-810-692
Témi Messmer - 21-826-870
Monika Multani - 18-061-663

Systems Group
Department of Computer Science
ETH Zurich
March 29, 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 MEMCACuHED_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:

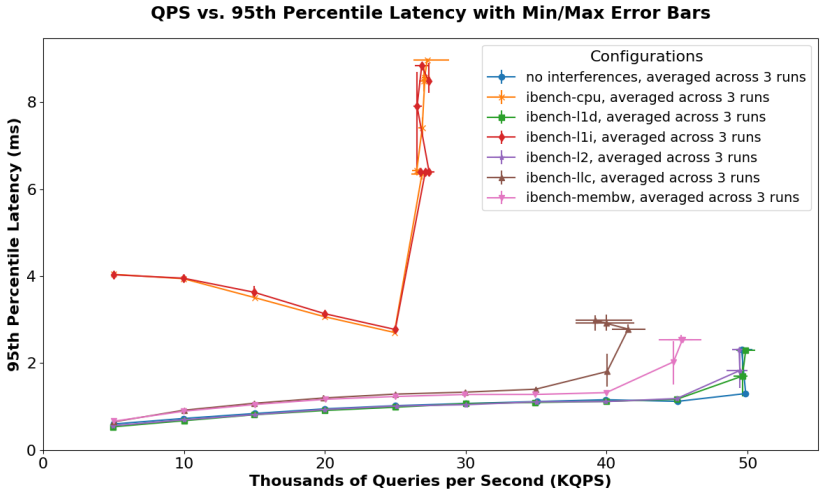

Figure 1: QPS vs. 95th Percentile Latency with Min/Max Error Bars

We increased the range of the y-axis to put all of our data in the graph, because our largest 95th percentile latency is larger than 8[ms]. Moreover, the Min/Max error bars were chosen because they offered the best readability.

(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`: The tail latency under CPU interference is significantly higher compared to other interference scenarios. This latency is indeed 3 to 4 times greater than the case with no interference, up to around 25K QPS, where we then observe an exponential increase. This indicates the saturation point at approximately 25K QPS, which is significantly lower than for other interference types, except for the L1i cache interference.

  Explanation: The significant impact of CPU interference on memcached's performance can come from its reliance on efficient CPU utilization for quick data access/manipulation. CPU interference diverts CPU resources to competing processes, and reduces memcached's CPU allocation, increasing its tail latency. This additional CPU demand lowers its saturation point because it struggles to manage various workloads. Also, the hypothesis of unused caches, as explained below, leading to increased CPU load, due to cache misses, could further explain the pronounced dependence on the CPU and impact of interference.

- `ibench-l1d`: With L1 data cache interference, the tail latency is very similar to when there is no interference. The same is true for the saturation point, which is around 50K QPS.

  Explanation: This lack of impact could be because memcached jobs naturally don't rely so much on this l1d cache due to a naturally low l1d hit rate (for example because of a too variable data access or a very big data working set). Data being evicted faster because of interference hence wouldn't be noticeable. Alternatively, memcached's resilience to L1d contention could be due to its handling of small/few key-value pairs that fit within the L1d cache even with interference. Lastly, the potential low penalty for L1d cache misses might mean that even if data is evicted due to interference, the overall impact remains limited, though this seems less probable.

- `ibench-l1i`: Similarly to cpu interference, the tail-latency is 3 to 4 times greater than the scenario with no interference, up to around 25K QPS, where we then observe an exponential increase in latency. The point of saturation is therefore around 25K QPS.

  Explanation: Interference in the L1i cache leads to increased latency due to pipeline stalls when instructions are delayed. This effect is noticeable in memcached due to its reliance on quick instruction execution. The early saturation point could be attributed to the L1i cache's limited size compared to memcached's working set, which becomes inadequate under higher loads/contention, resulting in more cache misses and slower instruction retrieval. Also, high contention can imply many context switches hence lli cache trashing and early saturation.

- `ibench-l2`: As for l1d, interference on the L2 cache doesn't have a significant negative impact on the tail latency and saturation point which is around 50K QPS, as with no interference.

  Explanation: The strongest hypothesis might that, as for the l1d cache, memcached typically handles quite random memory accesses (big data working set) that might not repeat so much, hence having a small hit ratio in the L2 cache, and therefore low dependency on this cache and not an earlier saturation point or higher latency when there is contention.

- `ibench-llc`: With llc interference, the tail latency under around 35K QPS is very similar to the configuration with no interference. However, after 35K QPS, the tail latency increases

such that the saturation point is around 40K QPS. We can also note that near the saturation point, the actual QPS the system can handle slightly regresses.

Explanation: A plausible hypothesis for the observed behavior under LLC interference, especially compared to L1d/L2 caches might be related to the LLC's capacity advantage. Being larger, the LLC is more likely to hold data that memcached reuses under high load in a no-interference scenario. With interference, especially under higher loads, this reusable and quite diverse data normally present in the llc may get evicted, forcing memcached to access the slower main memory, thus increasing tail latency, compared to contention on the L1/L2 caches. The earlier saturation point indicates memcached's increased data retrieval challenges and LLC's reduced efficiency under interference, as higher demand for diverse data forces reliance on slower main memory. Finally, the mentioned regression could come from the natural variance of the measurements or by increased queuing effects near the saturation point.

- `ibench-membw`: The tail latency under around 40K QPS is very similar to the configuration with no interference. However, after 40K QPS up to around 48K QPS, the tail latency increases such that the saturation point is around 45K QPS.

  Explanation: As explained previously, LLC cache usually holds much of memcached's data, reducing memory access and bandwidth interference's impact. Yet, under high loads, increased cache misses, even at LLC, due to varied data access heighten reliance on memory bandwidth, increasing data fetch times and raising tail latency. This scenario, coupled with bandwidth contention, leads to queuing at the memory controller, increasing memcached's response times and bringing the saturation point forward under high load/interference.

(c) **[2 points]**

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

  Answer:
  The `taskset` command is used to control the CPU cores on which containers run. Looking at the precise content of the YAML files, `taskset` makes sure that memcached's container should run on the same core as the benchmarks for cpu, l1d, l1i and l2 while the benchmarks for membw and llc run on a different core.
- 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: The goal of locating benchmarks on different cores is to be able to apply and measure interference effects targeted on a precise resource while not affecting other resources if possible. To apply interference, CPU, L1 caches, and L2 cache benchmarks need to share a core with memcached due to the core-local nature of these ressources (they are associated to a specific core). In contrast, L3 cache and memory bandwidth benchmarks are run on separate cores, as these resources are shared across cores, allowing for targeted testing without affecting memcached's other resources.

(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: Looking at our graph, apart from the CPU and L1i interference, all the curves and their y-axis confidence interval are below 2ms at a QPS of 35K. Therefore, the L2, l1d, llc and memory bandwidth interference can be safely collocated.

(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 project experiment is a closed system because there is a fixed number of clients that wait for their requests to be processed before sending new ones.
- What is the number of clients in the system? How did you find this number?
  Answer:
  In the `mcperf` command below, we can see that the '`-C 4`' option specifies that there should be 4 client connections created per thread. Since there are 16 threads, it makes a total of 4*16=64 clients.

```
$ ./mcperf -s MEMCACHED_IP -a INTERNAL_AGENT_IP \
--noload -T 16 -C 4 -D 4 -Q 1000 -c 4 -t 5 -w 2 \
--scan 5000:55000:5000,
```

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

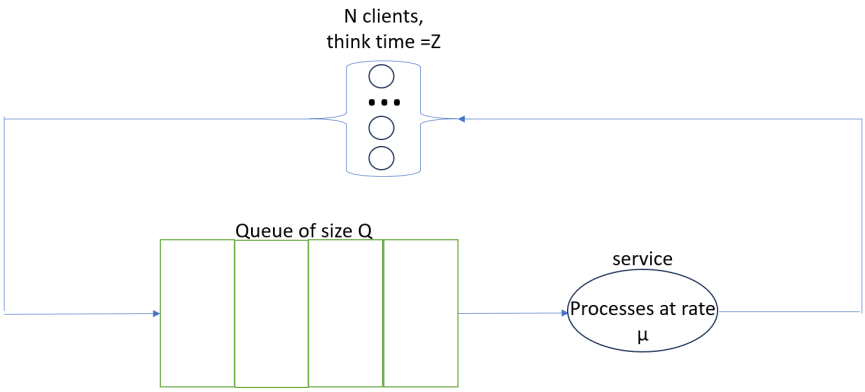

Figure 2: Modelling of the closed system

The diagram illustrates a closed system for memcached. There is a finite number of clients (N) and each one of them submits a request, which joins a queue of size Q. The memcached server then processes requests at a service rate of μ, and after their request is completed, clients think for a time Z before submitting a new one, showing the closed nature of the system. This described cycle repeats continuously.

- 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:
  With Little's law, the average response time (=average latency) of this system is $R = \frac{N}{X} - Z$. N is the number of jobs in the system, equal to the number of clients in a closed system, that is 64 here. X, the throughput, defined as the actual rate at which the entire system is processing requests is equal to the actual QPS achieved in our experiment. Z is the think time, that is the time a client spends after receiving feedback that its previous request has been processed by memcached and before sending a new one.

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	1.0	1.31	1.32	1.55	1.35	1.48	1.34
canneal	1.0	1.11	1.18	1.25	1.2	1.81	1.26
dedup	1.0	1.64	1.29	2.06	1.24	2.04	2.24
ferret	1.0	1.86	1.24	2.23	1.02	2.63	1.97
freqmine	1.0	2.04	1.04	2.02	1.03	1.77	1.62
vips	1.0	1.66	1.64	1.77	1.63	1.77	1.64
radix	1.0	0.96	1.01	1.15	1.01	1.36	0.99

   (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:` It seems to depend moderately on all resources but seems a little more sensitive to the l1i and llc caches. *Explanation:*
     Blackscholes' notable sensitivity to the LLC cache can be expected from the costly memory fetches required upon cache misses. Its reliance on CPU and L1i cache suggests a need for frequent, rapid floating point operations done with a compact instruction working set fitting within the L1i cache. This efficient fitting in the l1i, however, may diminish under contention. It seems to rely on the caches L1d and L2, probably because of its quite small working set implying data reuse.
   - `canneal:` It seems to depend a little on all resources except for the llc, which is more sensitive. *Explanation:*
     Canneal's high sensitivity to LLC interference suggests it handles large/irregular datasets, exceeding L1 and L2 capacities and depending on LLC's larger storage. Frequent LLC use and subsequent memory accesses during cache misses likely slow execution, explaining this resource requirement. Its minimal CPU/l1i reliance could indicate fewer/simpler operations or a load lying more toward managing memory.
   - `dedup:` It seems to depend on the CPU, but more importantly on the l1i, llc and memory bandwidth. *Explanation:*
     The dependency on the CPU might show this workload's need to perform computationally intensive tasks like hashing. These complex computations could come from executing repetitive algorithms, explaining the requirement on the l1i cache. The LLC might be relevant here because of the manipulation of large datasets exceeding

the capacity of L1d and L2 caches. Finally, the memory bandwitdh requirement could be linked to those big datasets that need to be both read and written back.

- `ferret:` It seems to depend to some extent on the CPU and the memory bandwidth, but more importantly to the l1i and llc. *Explanation:*
  The reasoning behind this can be similar to dedup, having complex repeating algorithms (like hashing/image similarity measures) involving large datasets that can't fit into upper caches. High memory bandwidth is needed to load and operate on the large database of image feature vectors this workload uses for similarity checks.
- `freqmine:` It seems to depend to some extent on the llc and the memory bandwidth, but more importantly on the CPU and l1i. *Explanation:*
  The strong dependency on the CPU shows that freqmine performs intensive computation for data mining like counting occurrences/identifying patterns. For the dependency on the l1i, this workload may involve hot loops and frequently called functions with high instruction locality. The dependency on the llc and memory bandwidth could be explained by a large working set and use of big data structures.
- `radix:` It does not seem to have strong resource requirements apart from very moderately on the l1i and moderately on the llc. *Explanation:*
  With a low CPU dependency and moderate l1i dependency, this workload may perform simple iterative operations. Same as before, the dependency on the LLC could come from handling relatively large datasets or data structures that fit only in the llc. Overall, this workload having low resource requirements might involve more cache-coherent and compute-efficient algorithms.
- `vips:` It has a profile very similar to that of blacksholes, but with a little bit more stress on each resource. *Explanation:*
  A similar reasoning to blackscholes can apply. However, each resource being a little more stressed, vips might involve more complex operations (like convolutions) and more repetitiveness of execution/access with larger datasets.

(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: Considering the conclusions made in part 1, memcached suffers a lot from CPU and l1i interference, moderately from llc and memory bandwidth interference and almost not from l1d and l2 interference. Therefore, the most suited jobs are canneal and radix, whose only default is to apply moderate pressure on the llc. The jobs blackscholes and vips could be considered but their pressure on the cpu and lli, even if moderate, could be enough to make memcached violate its SLO. Finally, dedup, ferret and freqmine should definitely be avoided because of the very high pressure they put on the l1i or cpu.

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: Shown in the 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:

- `blackscholes:`
  Overall, it is mostly sub-linear and in detail, it appears to scale almost linearly with a slope of 0.6 up to 4 threads before scaling sub-linearly. For the significance of the speedup, it seems reasonable given the graph to classify the speedup as significant if each new tread leads to an increase of more than 0.5 in the speedup, hence blackscholes has significant scaling up to 4 cores, before it becomes insignificant.
- `canneal:`
  We can draw similar conclusions as blackscholes : overall, it is mostly sub-linear and in detail, it appears to scale almost linearly with a slope of 0.5 up to 4 threads before scaling sub-linearly. Based on the significance definition stated for blackscholes, it significant scales up to 4 cores, before it becomes insignificant.
- `dedup:`
  It clearly scales sub-linearly. Based on the significance definition stated for blackscholes, it significant sales up to 2 cores, before it becomes insignificant.
- `ferret:`
  Overall, it is mostly sub-linear, and in detail, it appears to scale almost linearly with a slope of 0.8 up to 4 threads before scaling sub-linearly. Based on the significance definition stated for blackscholes, it significantly scales up to 4 cores (or 6 if we trust the interpolation between 4 and 8 threads), before it becomes insignificant.
- `freqmine:`
  This workload is the closest to overall linear scalability, with a slope around 0.6. With the proposed significance definition, it significantly scales up to 8 cores.
- `radix:`
  This workload shows a pattern similar to that of ferret but with a higher scaling initially, meaning it is overall sub-linear and in detail, it appears to scale linearly with a perfect slope of 1 up to 4 threads before scaling sub-linearly. Based on the significance definition stated for blackscholes, it significantly scales up to 8 cores.
- `vips:`
  Similarly to freqmine, this workload is overall sub-linear and in detail, it appears to almost scale linearly with a slope of 0.9 up to 4 threads before scaling sub-linearly. Based on the significance definition stated for blackscholes, it significantly scales up to 8 cores, 8 cores being barely included.

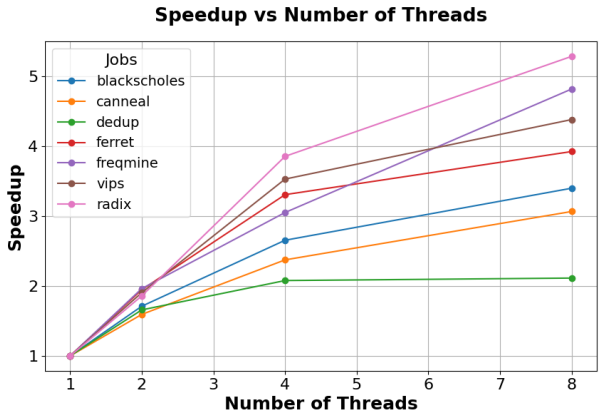

Figure 3: Speedup vs number of threads on different jobs using a single 8 CPU VM