Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Cloud Computing Architecture

Semester project report

Group 88
Zhiyi Hu - 23-943-285
Yuan Qin - 23-955-750
Guanshujie Fu - 23-955-560

Systems Group
Department of Computer Science
ETH Zurich
March 28, 2024

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`):

(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 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: For better readability, we average the last six overlapping data points for the benchmarks of CPU and L1i interference and merge them into one point for each benchmark.

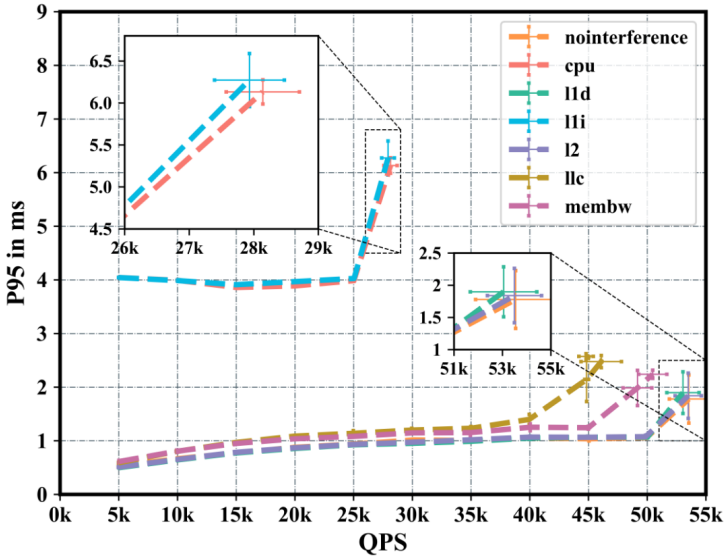

Figure 1: P95 vs QPS

(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:

- `ibench-cpu`: As we can observe from Figure 1, the CPU interference is one of the most influential interference and significantly degrades the performance of Memcached: **1)** The tail latency increases to 4ms when we introduce the CPU interference with QPS below 25k. There is a noticeable tail latency increase between QPS 25k and 30k. **2)** When we increase target QPS from 30k (saturation point) to 55k, the actually achieved QPS is saturated to around 28k to 29k.

  Explanation: This might be due to the multithreaded nature of Memcached for continuous processing of massive requests and data, which makes Memcached CPU-intensive.

- `ibench-l1d`: The L1 data cache interference shows negligible performance degradation compared with non-interference. As shown in above, the performances are almost the same even under high QPS.

  Explanation: Memcached might barely relies on L1d cache due to its distributed memory design. It needs to handle large scale data while L1d has small size that is infeasible for its use.

- `ibench-l1i`: The influence of L1 instruction cache interference is almost as significant as CPU interference.

  Explanation: The noticeable performance degradation of Memcached with L1i cache interference indicates that Memcached extensively uses L1i cache resources. This extensive usage likely stems from Memcached's design to continuously processing incoming requests. In this case, Memcached needs to execute numerous repetitive instructions and relies on the L1i cache to efficiently access instructions.

- `ibench-l2`: Similar to L1d cache. The L2 cache interference also presents negligible influences on Memcached performance.

  Explanation: Similar to L1d cache.

- `ibench-llc`: An observable degradation in performance of Memcached occurs with the LLC interference under high QPS: the tail latency increases significantly when we increase target QPS from 40k to 55k. The actually achieved throughput is saturated to around 45k to 48k when we increase target QPS from 45k (saturation point) to 55k.

  Explanation: The impact of LLC interference is larger than L1d and L2 interference because the size of LLC is larger and Memcached might rely more on LLC to handle large scale data. Under high QPS, data is evicted more frequently from LLC as the intensity of the LLC interference increases, leading to higher miss ratio. Hence, Memcached has to increase the memory accessing frequency, which is time consuming and degrades the performance.

- `ibench-membw`: The memory bandwidth interference also causes observable performance degradation under high QPS, but less significant than LLC interference: the tail latency varies slightly with QPS lower than 40k, but increases significantly when QPS increases from 40k to 50k. The throughput saturates to around 48k to 50k when the target QPS increases from 50k to 55k.

  Explanation: Memory bandwidth interference causes slightly less performance degradation than LLC interference. This may be because LLC hit ratio for Memcached is quite high by design and the data access of Memcached relies more on LLC than memory.

(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 set up the CPU affinity for processes. In the iBench benchmark configuration, **ibench-cpu, ibench-l1d, ibench-l1i, ibench-l2** are set to operate in the same core with **memcached**, while **ibench-llc** and **ibench-membw** are set in 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: We set the iBench benchmark configuration as above because we need to guarantee the *CPU, L1 cache* and *L2 cache* resources are shared between memcached and benchmark processes in the same core to make interference take effects. For *LLC* and *Memory BW*, they are shared between cores and hence not necessary to process in the same core.

(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]: As we can observe from Figure 1, **ibench-l1d** and **ibench-l2** should be able to safely collocate with **memcache** to guarantee a 95th percentile latency less than 2ms at 35K QPS. Because with any of these 2 interferences involved, the performance shows negligible variances.

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

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

  Answer: The system operates as a closed system due to two reasons: 1) From the commands we use to launch queries, we can observe that there is one mcperf agent with 16 threads and we establish 4 connections for each thread. If we consider each connection as a client, there are **64 clients** in the system and such a limited set of clients generating loads implies that the system is a **closed system**. 2) For every connection established, mcperf ensures to receive a response before sending the next request and measures the latency one at a time. This mechanism limits the actual QPS and throughput when the target QPS of the client exceeds the request handling ability of the server. Consequently, the system is **closed** because clients wait for response before sending next response.

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

  Answer: As discussed above, we have $16 \times 4 = 64$ connections between the agent and the server. Each client thread can send a request per connection and waits for the corresponding response to trigger the next request. Hence we can consider each connection as a client and there are **64 clients** in the system

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

  Answer: (See Figure 2 next page) We have 16 client threads and 4 server threads in the system. With each client thread establishing a connection with each server thread, every server thread receives and processes requests from all 16 client threads without interfering each other. So, we can consider the system as a closed system with 4 parallel servers and each server has its own queue to receive and process jobs from 16 clients.

- 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: Let λ denote the target QPS and μ denote the maximum number of jobs that the Memcached system can handle per second (The saturated throughput in the experiment), and R denote the average response time (The average latency in the experiment). When $\lambda > \mu$, all the servers shown in Figure 2 have all their 16 clients' requests in the system, with one being processed and others waiting in the queue. As

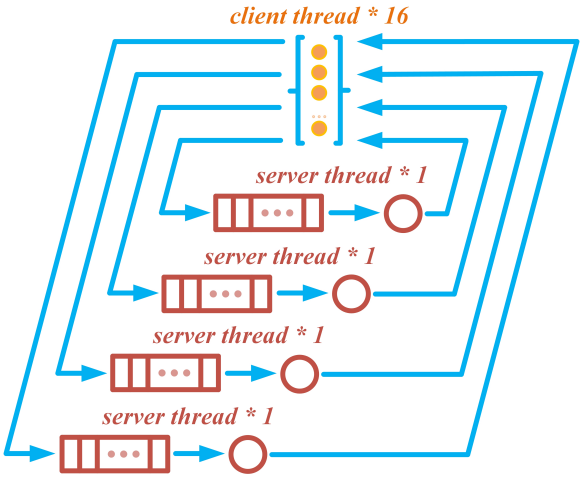

Figure 2: Diagram of the queuing system

all the clients wait for the response from the server to send the next request and $\lambda > \mu$, the client instantly sends the next request when it receives the response to approach its target QPS. Consequently, when the system is stable, each incoming request from client is the 16th job in the system. As the number of server threads is $m = 4$ the number of jobs in the system $N = 16$ and for its response time R, we have: $R = N/\frac{\mu}{m} = \frac{64}{\mu}$ *second* .

When $\lambda < \mu$, the server can handle jobs coming from clients and the target QPS can be approached by the actual achieved QPS. As λ denotes the target QPS, which is the number of jobs released per second, the client releases one job for one server thread during every time interval $T = \frac{1}{\lambda}$. So in the server side, all 16 incoming jobs can be processed during the time interval T and an incoming job can non-deterministically be the 1st, 2nd, ..., 16th job in the system. So, for the average response time R, we have: $R = \frac{N+1}{2}/\frac{\mu}{m} = \frac{34}{\mu}$ *second* .

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 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. Briefly summarize in a paragraph the resource interference sensitivity of each batch job.

Workload	none	cpu	l1d	l1i	l2	llc	memBW
blackscholes	1.00	1.30	1.34	1.53	1.29	1.43	1.52
canneal	1.00	1.25	1.35	1.46	1.35	2.26	1.42
dedup	1.00	1.60	1.25	1.96	1.21	2.04	1.67
ferret	1.00	1.82	1.17	2.13	1.17	2.68	1.95
freqmine	1.00	2.37	1.03	2.05	1.03	1.77	1.59
radix	1.00	1.10	1.12	1.18	1.11	1.55	1.11
vips	1.00	1.53	1.52	1.67	1.50	1.68	1.50

(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 resource interferences show a moderate effect on it. It is a bit more sensitive to L1 i-cache, LLC and memory bandwidth.
  Explanation: According to PARSEC, **blackscholes** is limited by floating-point calculation ability of CPU, which highly relies on L1/L2/L3 cache. Also, due to its PDE solver nature, it requires efficient memory access for large data sets.
- `canneal:` It is greatly influenced by LLC. L1d/L1i/L2 cache and *memory bandwidth* also present noticeable effect.
  Explanation: **canneal** utilizes cache-aware simulated annealing, which hence highly relies on cache usage for efficient execution.
- `dedup:` It is sensitive to L1i/LLC usage. CPU and memory bandiwdth usage also present influences while L1d/L2 show the slightest effect.
  Explanation: **dedup** is a high-ratio compression algorithm, which requires memory bandwidth for data accessing and CPU resources for pipelined computation.
- `ferret:` It is significantly influenced by LLC/L1i, while CPU and memory bandwidth indicate noticeable effects.
  Explanation: **ferret** is used for content-based similarity search, involving complex data processing that requires fast instruction access and efficient memory hierarchy utilization.
- `freqmine:` It degrades fast under high CPU and L1i usage. LLC and memory bandwidth also show noticeable effect, while L1d and L2 indicate slight effect.
  Explanation: **freqmine** employs an array-based data mining approach, which requires significant CPU resources to analyze data sets, and efficient L1 instruction cache to optimize performance.
- `radix:` It only shows moderate sensitivity to LLC usage and present resilience to other interferences.
  Explanation: Given **radix**'s nature as a sorting algorithm, it likely employs efficient, localized data processing techniques that mitigate the impact of resource contention.
- `vips:` **vips** indicates moderate sensitivity to all resources and present similar performance degradation for all interferences.
  Explanation: As an media processing application, **vips** requires substantial computational resources with efficient memory and cache to handle large images.

(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: `radix` appears to be a good candidate. Memcached's latency spikes under CPU and L1i interference, underscoring their impact on its performance. Only `radix` displays minor sensitivity to both the interferences, implying lower resource contention.

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:

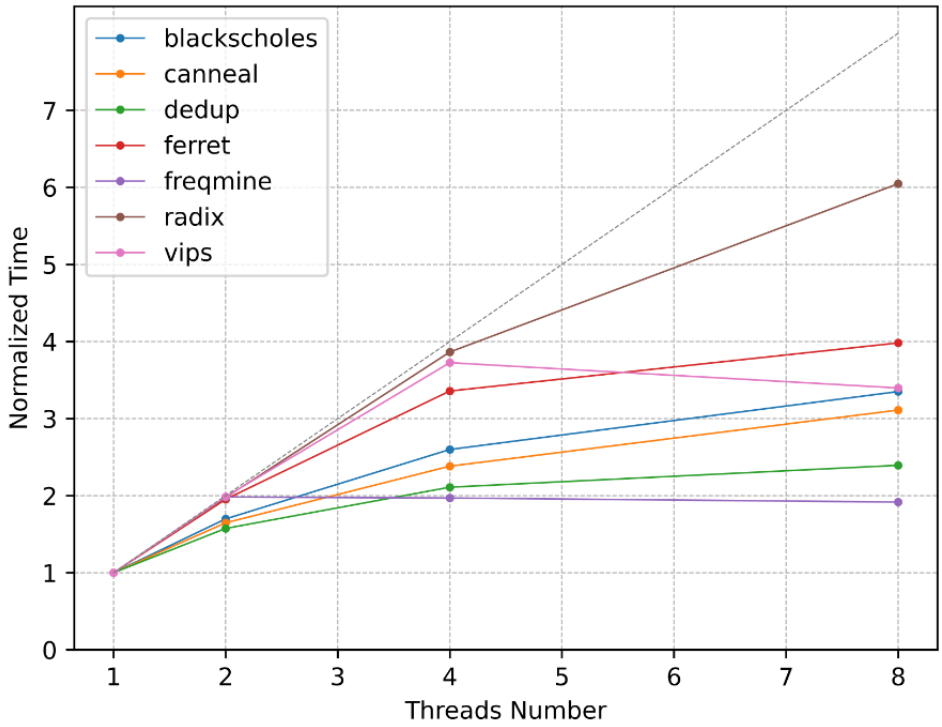

Figure 3: Normalized Time vs Threads Number

(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 consider a performance enhancement greater than 50% as significant. Under this circumstance, we do the following analysis:

- `blackscholes:` it shows sub-linear scalability for all threads number. The speedup is significant under 2/4 threads, and indicates slight enhancement with 8 threads.
- `canneal:` it shows sub-linear scalability for all threads number. Close to **blackscholes**, the speedup is significant under 2/4 threads and trivial under 8 threads.
- `dedup:` it shows sub-linear scalability for all threads number, and the performance speedup is only significant under 2 threads.
- `ferret:` it presents linear scalability with 2 and 4 threads, and sub-linear scalability with 8 threads. The performance enhances significantly with 2/4 threads.
- `freqmine:` it presents linear scalability with 2 threads, and almost no enhancement with 4/8 threads.
- `radix:` it presents linear scalability with 2 and 4 threads, and sub-linear scalability with 8 threads. **radix** shows significant speedup with more threads as increasing threads number indicates more than 50% speedup compared to previous one.
- `vips:` it presents linear scalability and significant speedup with 2 and 4 threads. However, the performance slightly degrades with 8 threads.