ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Systems@ETH Zürich

Cloud Computing Architecture

Semester project report

Systems Group
Department of Computer Science
ETH Zurich

Instructions

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

Part 1 [20 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 80000 in increments of 5000:

```
$ ./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 5000:80000:5000
```

Repeat the run for each of the 7 configurations (without interference, and the 6 interference types) **at least three times** (three 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.

(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 80K). (note: the actual achieved QPS, not the target QPS)
- 95th percentile latency on the y-axis (the y-axis should range from 0 to 10 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.

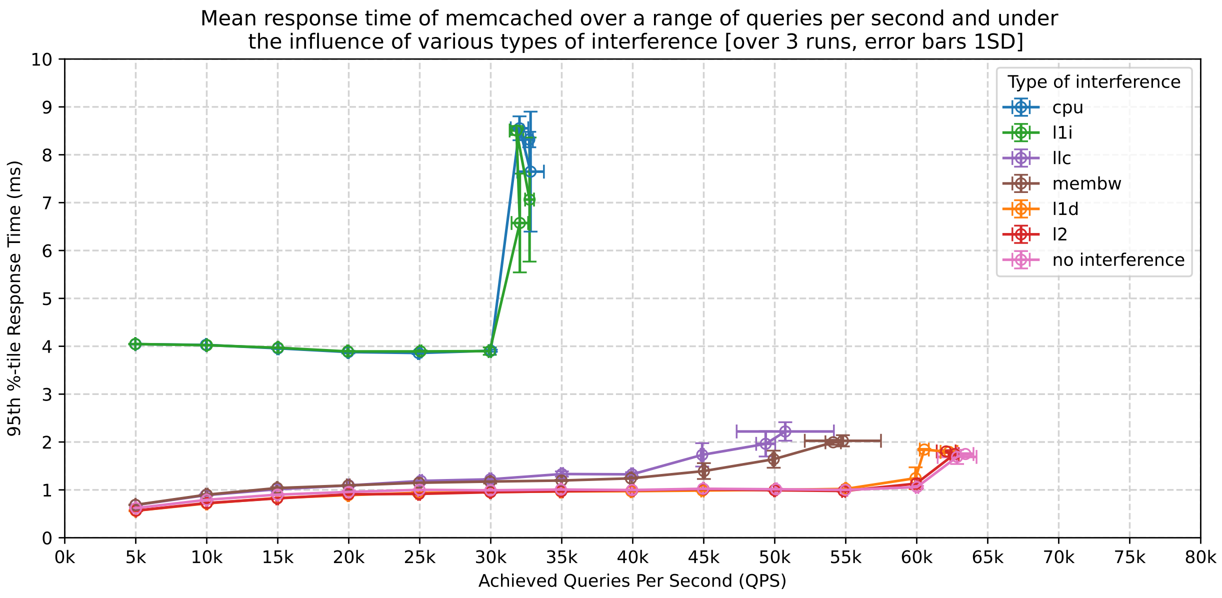

Figure 1: 95th %-tile Response Time of different interferences at memcached, a latency sensible application. The target queries per second ranged between 5,000 and 80,000.

(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.

First of all, it is important to observe that the achieved queries per second are analogous to throughput. `cpu` and `l1i` are the ones that underperform the most the memcached. Even for 5,000 QPS the response time increases up to 4 ms and when they reach around 30k QPS (the elbow) the response time sharply increases. This is similar to the behavior in the interactive law between throughput and response time that we saw in the lecture. This is likely because the tight packing of jobs due to high user loads impedes the system's ability to perform important overhead operations and thus slows down the response time drastically. This slowdown further inhibits the execution of these overhead operations, which leads to the exponential nature of response time growth. Because these overheads, e.g. scheduling, as well as the processing of the queries themselves all rely on the CPU, it may be reasonable to argue that it is more susceptible than other system components to reach this point of exponential negative feedback, thus explaining why `cpu` is one of the only interference patterns that cause this.

The other source with this behavior, `l1i`, is the l1 cache for CPU instructions. It is natural to hypothesize that the CPU is heavily reliant on this cache and would therefore suffer in performance similarly if it were interfered with.

The other interference patterns have a noticeably different response time curve compared to the previous two interference. We suspect that this is due to the system being in a *saturated* but not *overloaded* state. As the user load (different from throughput) increases, the response time begins to grow linearly. However, this growth worsens at an exponential rate once we reach a certain "knee in the curve" at which the system becomes overloaded. Since our response time graph (from the data we collected) does not begin to grow exponentially for these interference patterns, we hypothesize that the query loads in our experiments were not sufficient to make the system become overloaded under these interferences. However, as we do see a small ramp up in response time (and therefore an elbow) it is likely that reaching 50k achieved QPS (for `llc` and `membw`) was predicated on a user load high enough to induce a saturated state in the system.

Similar analysis applies for the last observed group, `l1d` and `l2`, and no interference which experience this elbow at around 60k achieved QPS. The interferences at L1 data and L2 does not significantly affect the performance. It may be due to the fact that there is always one per core and the latency is lower compared to other caches. The variability is higher in the achieved QPS than in the response time, which may be a sign that the system is overloaded rather than saturated. In the case of the `cpu` and `l1i` the response time around the "knee in the curve" varies substantially, between 6 and 9 ms. This could be a sign that the respective user load is in an overloaded state where problems in the system arise.

We believe the difference in variance (shown by the horizontal error bars) in throughput between `cpu` (and `l1i`) relative to the rest of the interference patterns lies in the distinction between saturated and overloaded states. When the system is overloaded, this indicates that the throughput has reached or is incredibly close to the theoretical maximum service rate. This means that it is impossible for there to be positive variance beyond this upper bound, and unlikely for there to be a significant negative variance due to the fact that the user load is large enough to induce this overloaded state to begin with. On the other hand, when the system is saturated but not yet overloaded, it is still possible for there to be both positive and negative variance.

(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 is used to set the CPU affinity of a process. It bonds a process to a given set of CPUs on the system. In this case, the container of the `memcached` is set to the same core as the iBenches of `cpu`, `l1d`, `l2` and `l1i`. `llc` and `membw` are set to another core. This is because different cores (with their respective L1 and L2 caches) share the same L3 cache, as well as the memory bandwidth is set to all different accesses to memory. Given the sharing of these sources among different cores, it is important to run them in a different core so that their interference don't affect the rest sources.

(d) [**2 points**] Assuming a service level objective (SLO) for memcached of up to 1.5 ms 95th percentile latency at 65K QPS, which iBench source of interference can safely be collocated with memcached without violating this SLO? Briefly explain your reasoning.

If we observed the obtained results, all of the iBench sources of interference produce a response time higher than 1.5 ms for any QPS. This means that no source of interference could be allocated to the memcached and maintaining this SLO at the same time. Strictly speaking, even with no interference we do not reach the SLO target.

Among the sources of interferences there are three clear groups regarding how they affect memcached. `cpu` and `l1i` effects are drastic, whereas `llc` and `membw` have a moderate effect. Finally `l1d` and `l2` slightly affect the performance. The closest option to reach this SLO would be to choose the last two sources, although it will not be clearly achieved based on our measures.

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

Workload	`none`	`cpu`	`l1d`	`l1i`	`l2`	`llc`	`memBW`
dedup	1.00	1.53	1.23	2.47	1.28	2.23	1.68
blackscholes	1.00	1.41	1.37	1.89	1.35	1.71	1.44
ferret	1.00	1.90	1.00	2.67	1.01	2.66	2.10
freqmine	1.00	1.94	0.99	1.96	0.98	1.82	1.55
canneal	1.00	1.26	1.32	1.59	1.29	2.07	1.45
fft	1.00	1.32	1.28	1.91	1.28	2.01	1.53

There are general patterns in resource interference sensitivity common to all batch jobs. Namely, the performance of all is significantly affected by interference in the last level cache (LLC) and level 1 instruction cache (L1I). Interference induced in memory bandwidth and CPU generally had a moderate effect on a given job's performance. Finally, the level 2 (L2) and level 1 data (L1D) caches affect it the least. The *freqmine* benchmark is mostly unaffected by interference in the L1 data and L2 caches, moderately affected by interference in memory bandwidth and more seriously impaired by interference in the L1 instruction cache, last level cache and CPU. The *ferret* benchmark is affected similarly to *freqmine* by most types of interference, but the effect of L1I, LLC and memory bandwidth are comparatively more severe.The *canneal* benchmark only appears to be seriously affected by Last Level Cache interference. Other types of interference produced a moderate effect. The *dedup* and *fft* benchmarks performed similarly to each other. Both are moderately affected by CPU and memory bandwidth interference and slightly less sensitive to L1 data and L2 caches. Both appear to be very sensitive to interference in the L1 instruction and last level cache, with the performance of *dedup* being affected comparatively more. Finally, the *blackscholes* benchmark suffered moderate performance decreases from all types of interference, with interference in the L1I and LLC affecting it slightly more.

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 1.5 ms P95 latency at 65K QPS?

The effect that the interference has on each batch job gives hints about the resources requirement of each application and how they are deployed. Most benchmarks share data intensely. *Blackscholes* is uniformly affected by all interferences. It shows a substantial amount of sharing, and the coarse granularity is susceptible to load imbalance. Since it is data parallel and the usage data is low, it is not affected significantly in `llc` and `l1i` as other jobs. *dedup* and *ferret* are most impacted by `l1i` and `llc` since they have high data utilization. They also use pipeline parallelization model and all data has to be passed from stage to stage.

In the middle we find *freqmine*, *canneal* and *fft*. `llc` is more affected in those cases where there is fine granularity as more accesses are required.

In Part 1 we commented that there are not any interference that could be collocated with memcached without violating this SLO. The required SLO is too demanding and it is even difficult to reach it in normal conditions. If we had to choose, a solution would be to look at those applications which are not sensitive to the most used resources in memcached. Since memcached is resource heavy in `CPU` and `L1i`, it would not work properly with applications that require a lot of these resources.

The best option to collocate would be *canneal*, which is the one least affected by `cpu`. Secondarily we could select *blackscholes* and *fft*. The worst options would be *ferret* or *dedup*, since they both are cpu intense, like memcached.

3. **[10 points]** Plot a single 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. Briefly discuss the scalability of each application: e.g., linear/sub-linear/super-linear. Do any of the applications gain a significant speedup with more threads? Explain what you consider to be "significant".

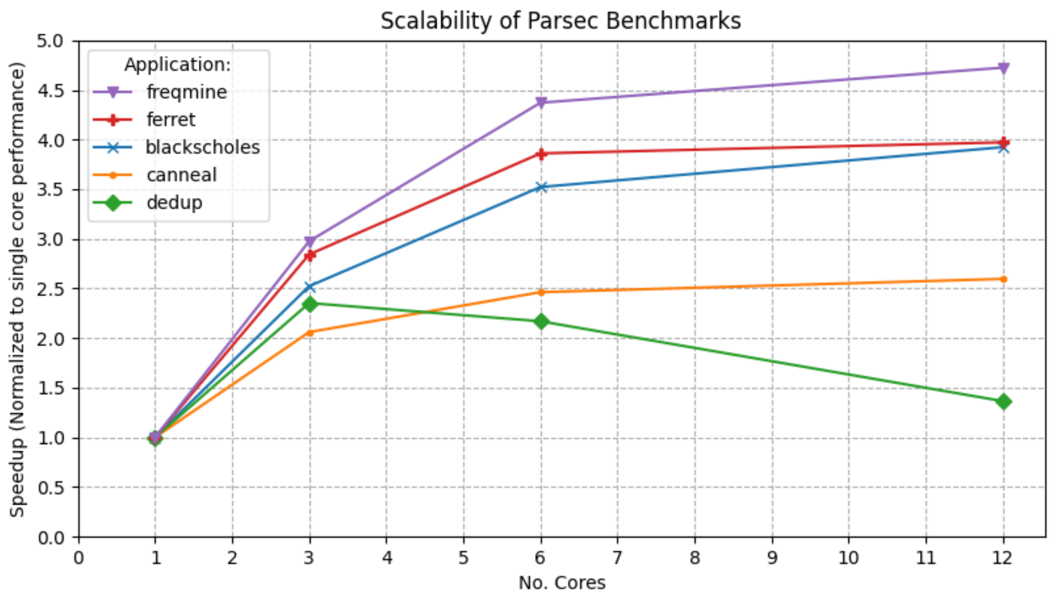

Figure 2: Speedup achieved by select benchmarks when increasing the number of cores on host machine.

To aid in further discussion, we will present our definition of a "significant" speedup first. Due to a lack of repeat measurements, it is impossible to accurately assess the variance in execution times at a given set of parameters. Consequently, there is a risk that small observed speedups are due to chance.

In the ideal case, we would expect an increase in computational power to engender a linear increase in performance. Furthermore, an increase in the number of cores by a factor of n should increase the speedup by the same factor. Note that what constitutes a great, average or even just significant speedup is application dependant, meaning our only point of reference is this ideal linear speedup.

Based on these factors we will aim to set a conservative threshold for what constitutes a “significant” speedup based on the speedup attained per added core. For this discussion, we will consider a speedup of 0.5 per added core to still be significant.

In practice, the ideal linear speedup is seldom attained, instead, we expect to see a diminishing return on investment for the addition of cores. This trend can be observed with all of the benchmarks tested, as all manage to achieve significant performance gains when moving from 1 to 3 cores which peter off at different rates. By the 12 core mark, none of the tested benchmarks is showing significant performance improvements compared to 6 cores. The performance of all tested benchmarks is therefore sub-linear. There are several reasons why this may be the case. Firstly, adding more compute power may cause a benchmark initially bottlenecked by computation to switch to being bottlenecked by some other resource (e.g. i/o or memory), and therefore stop improving. Secondly, since only a certain fraction of an application is parallelizable, the maximum speedup achieved is limited from above. For example, if only half of an application’s code is parallelizable, we can expect the addition of cores to decrease its runtime by at most a factor of two (this follows from Amdhals law).

Despite exhibiting similar behaviour overall, there are significant differences between benchmarks in terms of how well they scale with the addition of cores and how many cores can be added before performance stops improving.

The best scaling is exhibited by the *freqmine* and *ferret* benchmarks. Up to the 3 core measurements, both of these are essentially scaling optimally. When moving from 3 to 6 cores, the achieved performance gains have noticeably slowed down, however, their overall performance remains the highest and they have continued to improve at least as well as other benchmarks. The *blackscholes* workload has also continued to improve significantly up to 6 cores, however, its initial performance gains at 3 cores are smaller than that of the other two. Although the *dedup* and *canneal* benchmarks show significant improvements when moving to 3 cores, they stop improving earlier than the other benchmarks. By the 6 core measurements, they have already failed to improve significantly.

Moving from 6 to 12 cores, none of the benchmarks displays significant performance improvements, indicating that their parallelization has become limited, likely because of one of the aforementioned factors.

The case of the *dedup* benchmark is particularly interesting, as moving to higher core counts appears to have decreased its performance [so called parallel slowdown]. This is typically due to the overhead of communications between core. When the communications overhead of an additional core exceeds the performance improvement given by that core, the overall performance gets worse and we observe a decrease in speedup.

To summarize, all applications achieve significant speedups from adding up to 3 cores. In the cases of *ferret*, *canneal* and *blackscholes*, significant scaling was observed for longer, up to the 6 core mark. Beyond this none of the benchmarks achieved significant scaling.