1. General Feedback to All Groups

We first give feedback on things that *many* of you did and perhaps could use some feedback on. Specifically:

**Overall Presentation**

Make your presentation uniform! Make slides that look nice and make them all look consistent and clean. Spend time on this!

**Data Presentation**

Label all graph axes! You should never present a graph without clear labels on all axes – it just makes it harder to understand.

Use the right sized units! e.g., don’t put .0001 seconds when you can simply put .1 ms and don’t put 1000000 microseconds when you can put 1 second. Figure out what the best unit is for the granularity of measurement and use that.

Show all the data if you can. Not just averages – averaging hides information! Or, at least, look at all the data before you do the average, etc.

Graphs: generally, start at zero on the y-axis. Especially for bar graphs. Otherwise you could be misrepresenting the data.

Make graphs next to each other have same height/max value for y axes! This allows comparison across graphs.

Put the thing you’re comparing on the same graph near one another. Graphs facilitate comparison; thus, put the things you want the audience to compare near one another.

Use consistent colors across graphs, e.g., GRPC always blue, Thrift always green. That way, it’s easier to read graphs!

**Empirical Methods**

In general, you cannot meaningfully compare time across machines. Thus, when measuring, you have to think about what you can do on one machine.

A disk seek is not just an `fseek`! To get a disk to do a seek, you have to think about how to avoid caches (like the file system page cache, or the disk’s cache) and make sure the disk is read/writing blocks from far away spots on the disk.

Compare your measured numbers to spec sheets. What does the CPU/disk/SSD/DRAM expect to deliver? How do your numbers compare? Are there major discrepancies?

Testing should be done on the same setup, to facilitate comparison! Do not compare one system on one machine to another system on a completely different machine. This is not meaningful!

If the number looks weird, run it again. Ask why the number looks weird. Is it repeatable? What could be going on? What other numbers can you get to figure out what is going on? You can remove noise with careful experimentation!

Be curious! Figure out why things are the way they are. For example, a number of you used the C++ `chrono::whatever` timer. How does it work? You can find out!
Drawing Conclusions

Show data, and then try to draw conclusions. What does the data show? Make sure the reader/audience can see the same data, and draw the same conclusions from the data you have shown.

Don’t make guesses as to why - you’re probably wrong! Make a guess so as to then do subsequent measurements, and confirm/deny your guess. Having a guess is easy; showing that your reasoning is solid is hard and requires work.

Don’t draw strong conclusions when no strong conclusion is warranted! It’s better to say “I don’t know” than to put forward the wrong conclusion.

2. Feedback to Your Group

Slides had good structure generally and there are lots of details. Good!

Part-1

- Good comparison for timers, detailed comparison of specification of timers.
- When testing the memory reference and cache reference, look at the CPU spec. How large is the L1/L2/L3 cache and give a workload to some reasonable size of array that could force your access comes from certain layer in this cache hierarchies. E.g., when testing memory reference, how to ensure no access comes from L1/L2...
- Branch mispredict:
  - One way is to use quicksort and give it a sorted array vs. not-sorted array, and save your observed time difference. Then the issue becomes how you could actually find number of mispredict. One way to do it is using Linux perf tool which supports monitoring HW events branch-misses.\(^1\)\(^2\)
  - \[ \text{cost mispredict} = \frac{\text{Time difference in sorting}}{\text{number of branch misses difference}} \]

Part-2

- A+ => Use OSTEP. So obviously the code looks familiar.
- Why use disk in a communication test? This doesn’t make sense really. It’s better to just send from memory to memory.
- It’s good to put code in slides, but skip during presentation when obvious

Part-3

- marshal/unmarshal timers
  - It’s good to insert code by checking the stacktrace and find the right place of marshalling.
  - The table have some odd numbers
    * E.g., 700->500->400 for int32. (Suspect it has something to do with caching effect, page cache or something related to your reading from disk...)
    * 5 times and get average? Generally assuming repeat for >10000 times for each marshalling get the time and then get average overhead.
  - O3 is worse? – This is very odd. Needs to have number that make more sense.
- The request/response time:
  - Testing the clock difference between machines? looks like buffer overflow
- Network bandwidth looks reasonable.
- It’s great to read the UNet paper and try to do some similar experiments. But their numbers are all explainable, right?
- The last graph is interesting but hard to interpret! log scale doesn’t make sense.

\(^1\) perf Examples
\(^2\) Linux Perf Programming
Handin

- Roster

<table>
<thead>
<tr>
<th>Member</th>
<th>Email</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wuh-Chwen Hwang</td>
<td><a href="mailto:wuh-chwen@cs.wisc.edu">wuh-chwen@cs.wisc.edu</a></td>
</tr>
<tr>
<td>Shichun Yu</td>
<td><a href="mailto:syu274@wisc.edu">syu274@wisc.edu</a></td>
</tr>
</tbody>
</table>

- Checked Handin: wuh-chwen
- Please submit slide somewhere (either canvas or handin).
- Thanks for listing “not tested” for part1.
- Quite clear table in README, looks good.
- A quite explainable README.