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

Nice title pic in the first slide.

Part-1

- `clock_gettime` used `steady_clock`?
- There is no disk numbers, but it’s fair that people in 2020s doesn’t know about HDD.
- SSD numbers seem too slow? (not 4 bytes 1000 times).
- Branch mispredict:
  - Using Linux perf event `<perf_event.h>` to find the number of branch misses, great.
  - The issue here is in the program between line 51-60, the code is quite deterministic such that CPU can predict the branches quite well. You get the number of misses, the total execution time, do you know how much time actually should be accounted into branch hit? Essentially you are using $\frac{Time(\text{Hit}) + Time(\text{Miss})}{\text{Number of Misses}}$.
  - One way to solve this is to create difference. E.g., sort an sorted array vs. unsorted array to ensure two run have different #of misses, and use the difference of time.

```c
47 ioctl(fd, PERF_EVENT_IOC_RESET, 0);
48 ioctl(fd, PERF_EVENT_IOC_ENABLE, 0);
49
50 int sum = 0;
51 clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &start);
52 for (i = 0; i <= ITS; ++i) {
53     if ((i & 1) == 0) {
54         sum += 1;
55     }
56     else {
57         sum -= 1;
58     }
59 }
60 clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &end);
61 ioctl(fd, PERF_EVENT_IOC_DISABLE, 0);
62 read(fd, &count_1, sizeof(long long));
63 diff_1 = BIL*(end.tv_sec - start.tv_sec) + end.tv_nsec - start.tv_nsec;
64 cout << " branch mispredicts in " << count_1 << " time " << diff_1 << endl;
```

Part-2

- Basic UDP lib stuff looks good.
- Overhead: while showing minimum makes sense (it’s one data point). Need to show average and median.
- Good drop graph. Some of the bandwidths look low?
Part-3

- Marshalling/Unmarshalling
  - Why you stop at 4K size? A general good idea for doing such experiment design is to some extreme data points. E.g., one maximal network packet size (per TCP).
  - Avoid stacking these numbers, hard to read (a bit confusing).
  - Complex structures: missing. It would be good to use some map/linked list (structures that have pointers).
  - Numbers look reasonable.
  - Thrift is slower.

- GRPC:
  - 4K number is the lowest! This makes little sense. A mystery!
    * What’s the specific latency? How does it compare with your UDP library basic latency in server-client case?

- Thrift:
  - Thrift numbers and the streaming numbers make more sense.

Handin

- Roster

<table>
<thead>
<tr>
<th>Member</th>
<th>Email</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wiley Corning</td>
<td><a href="mailto:wcorning@wisc.edu">wcorning@wisc.edu</a></td>
</tr>
<tr>
<td>Samuel Kruse</td>
<td><a href="mailto:sdkruse@wisc.edu">sdkruse@wisc.edu</a></td>
</tr>
<tr>
<td>Hemalkumar Patel</td>
<td><a href="mailto:hpatel5@wisc.edu">hpatel5@wisc.edu</a></td>
</tr>
</tbody>
</table>

- Checked Handin: skruse
- It’s nice to use ipynb for the visualization.
- YOUR TEAM format is wrong which could break our script.
- Sent an updated README? ok, accepted. But please ensure to follow the instruction for time carefully, as well as the format strictly next time.