6.033 in the news

Meta has built an AI supercomputer it says will be world’s fastest by end of 2022

*Designed to train the next generation of machine learning systems*

By James Vincent  |  Jan 24, 2022, 12:00pm EST

Sorensen offers one extra word of caution, too. As is often the case with the “speeds and feeds” approach to assessing hardware, vaunted top speeds are not always representative. “HPC vendors typically quote performance numbers that indicate the absolute fastest their machine can run. We call that the theoretical peak performance,” says Sorensen. “However, the real measure of a good system design is one that can run fast on the jobs they are designed to do. Indeed, it is not uncommon for some HPCs to achieve less than 25 percent of their so-called peak performance when running real-world applications.”

In other words: the true utility of supercomputers is to be found in the work they do, not their theoretical peak performance. For Meta, that work means building moderation systems at a time when trust in the company is at an all-time low and means creating a new computing platform — whether based on augmented reality glasses or the metaverse — that it can dominate in the face of rivals like Google, Microsoft, and Apple. An AI supercomputer offers the company raw power, but Meta still needs to find the winning strategy on its own.

6.033 Spring 2022

Lecture #7: Performance + Other Concerns

performance, with a deep dive into storage and filesystems
operating systems enforce modularity on a single machine using virtualization in order to enforce modularity and have an effective operating system, a few things need to happen.

1. programs shouldn’t be able to refer to (and corrupt) each others’ memory
2. programs should be able to communicate with each other
3. programs should be able to share a CPU without one program halting the progress of the others

you’ve also seen virtualization as a technique for running multiple operating systems on the same physical hardware.
operating systems enforce modularity on a single machine using virtualization in order to enforce modularity + have an effective operating system, a few things need to happen

1. programs shouldn’t be able to refer to (and corrupt) each others’ memory

2. programs should be able to communicate with each other

3. programs should be able to share a CPU without one program halting the progress of the others

**today:** performance more generally, with a focus on storage, and how the abstractions that an operating system provides impact our systems
performance issues have influenced a lot of the system designs you’ve seen so far

**latency**: how long does it take to complete a single request?

example: how long does it take to retrieve a particular piece of memory in an OS?

**throughput**: how many requests per unit of time?

example: how many reads or writes can a system do to a disk at once?

**utilization**: what fraction of resources are being utilized? this puts our performance measurements in context

often we see latency remain low until the system is heavily utilized, and then it rises.

throughput, on the other hand, increases as requests increase, and maxes out when the system is fully utilized.
Performance issues have influenced a lot of the system designs you’ve seen so far.

**Latency:** How long does it take to complete a single request?

Example: How long does it take to retrieve a particular piece of memory in an OS?

**Throughput:** How many requests per unit of time?

Example: How many reads or writes can a system do to a disk at once?

**Utilization:** What fraction of resources are being utilized? This puts our performance measurements in context.

Our general approach to improving performance is to measure our systems to find a bottleneck, and then to relax the bottleneck with general techniques such as caching, parallelism, etc.

We’ll make this concrete with an example: performance in reading/writing to a file.
the disk is often the main bottleneck in reading/writing stored data.
The disk is often the main bottleneck in reading/writing stored data. Hard disk drives (HDDs) are common in datacenters.

Example HDD specs (Hitachi 7K400):
- Capacity: 400GB
- Number of platters: 5
- Number of heads: 10
- Number of sectors per track: 567-1170
- Number of bytes per sector: 512
- Time for one revolution: 8.3ms
- Average read seek time: 8.2ms
- Average write seek time: 9.2ms

Since so much time of reading/writing is spent seeking, avoiding random access can improve performance.
the disk is often the main bottleneck in reading/writing stored data

since SSDs don’t involve moving parts, disk seeks are not a concern (this is one of the reasons SSDs are so much faster than HDDs)

however, because of how writes are done, the SSD controller is careful about how it writes new data and makes changes to existing data

solid state drives (SSDs)
common in personal computers
the **disk** is often the main bottleneck in reading/writing stored data

**hard disk drives** (HDDs)  
common in datacenters

**solid state drives** (SSDs)  
common in personal computers
so far, we have always imagined reading/writing data via the abstraction of a **filesystem**. does that abstraction ever get in the way?

- Platters
- Tracks
- Sectors

**Hard disk drives** (HDDs)

- Common in datacenters

**Solid state drives** (SSDs)

- Common in personal computers
so far, we have always imagined reading/writing data via the abstraction of a filesystem. does that abstraction ever get in the way?

how should the data be stored as files?
one file for everything? one file per table? per row? per column? per cell?
so far, we have always imagined reading/writing data via the abstraction of a filesystem. does that abstraction ever get in the way?

“what’s Katrina’s office number?”

“how many recitation instructors have offices in building 32?”

DBMS

<table>
<thead>
<tr>
<th>first name</th>
<th>last name</th>
<th>building</th>
<th>room</th>
</tr>
</thead>
<tbody>
<tr>
<td>katrina</td>
<td>lacurts</td>
<td>38</td>
<td>476</td>
</tr>
<tr>
<td>karen</td>
<td>sollins</td>
<td>32</td>
<td>G534</td>
</tr>
<tr>
<td>mike</td>
<td>cafarella</td>
<td>32</td>
<td>G887</td>
</tr>
</tbody>
</table>
...

<table>
<thead>
<tr>
<th>first name</th>
<th>last name</th>
<th>role</th>
</tr>
</thead>
<tbody>
<tr>
<td>katrina</td>
<td>lacurts</td>
<td>lecture</td>
</tr>
<tr>
<td>karen</td>
<td>sollins</td>
<td>recitation</td>
</tr>
<tr>
<td>mike</td>
<td>cafarella</td>
<td>recitation</td>
</tr>
</tbody>
</table>
...

how do standard caching policies behave here?
is least-recently-used the best policy?
so far, we have always imagined reading/writing data via the abstraction of a *filesystem*. does that abstraction ever get in the way?

"what’s Katrina’s office number?"

"how many recitation instructors have offices in building 32?"

<table>
<thead>
<tr>
<th>first name</th>
<th>last name</th>
<th>building</th>
<th>room</th>
</tr>
</thead>
<tbody>
<tr>
<td>katrina</td>
<td>lacurts</td>
<td>38</td>
<td>476</td>
</tr>
<tr>
<td>karen</td>
<td>sollins</td>
<td>32</td>
<td>G534</td>
</tr>
<tr>
<td>mike</td>
<td>cafarella</td>
<td>32</td>
<td>G887</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>first name</th>
<th>last name</th>
<th>role</th>
</tr>
</thead>
<tbody>
<tr>
<td>katrina</td>
<td>lacurts</td>
<td>lecture</td>
</tr>
<tr>
<td>karen</td>
<td>sollins</td>
<td>recitation</td>
</tr>
<tr>
<td>mike</td>
<td>cafarella</td>
<td>recitation</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

the DBMS knows so much about the data and related queries that it can do a very good job at predicting which byte it needs next. it’s in a good position to exploit block-level control over loading or evicting data to memory.
Performance is important throughout systems. We often measure throughput, latency, and utilization, and use techniques such as caching and batching to improve performance.

In reading/writing files, the disk is often the bottleneck. Performance changes dramatically depending on the pattern of reads/writes (e.g., random vs. sequential access).

Abstractions such as the filesystem work well in many places, but sometimes get in the way, especially when it comes to performance. Block-level control can make sense for certain applications, such as databases.

Block-level control isn’t perfect for every type of database; some do just fine with filesystems.