CSE 291 Virtualization: Course Summary

Yiying Zhang
Outline

• Course summary
• Brief intro to major datacenters/clouds
• Hints for computer systems design

• Final project presentation this Wed
  • (4min presentation + 1min Q&A) * 17 groups
• Final project summary report due 12/9
• Course evaluation!
Virtualization Approaches

- Hosted interpretation
  - Interpret each instruction, super slow (e.g., Virtual PC on Mac)

- Direct execution with trap-and-emulate
  - Requires a virtualizable processor and only works for the same architecture

- Direct execution with binary translation
  - Works with non-virtualizable processor, but implementing VMM is tricky

- Direct execution with hardware-assisted virtualization
  - Needs new generation of hardware (which is the norm now), mode switching is still not optimized

- Direct execution with paravirtualization
  - Good performance and works with non-virtualizable processors, but require guest OS changes

- OS-level virtualization, library-level, language (app)-level, unikernels, etc.
  - More lightweight and faster to start, but less secure
Virtual Machine Architectures

1. Xen / VMware ESX
2. VMware Workstation / VirtualBox
3. Linux KVM
Major Clouds and Datacenters
AWS

• Biggest market share, longest history
• Highest compute (and other service) options
  >= 136 instance types in 26 families
• Storage
  – Simple Storage Service (S3)
  – Elastic File Service (EFS) / Elastic Block Service (EBS)
• Many other services
  – Lambda (serverless)
  – ECS/EKS (managed containers)
  – …
Amazon

- **Storage**
  - Dynamo, S3, EFS, EBS

- **Database/NoSQL**
  - DynamoDB, Redshift, ElastiCache

- **Network**
  - Customized NICs, virtualization support

- **Hardware**
  - ASIC (Nitro), x86, ARM

- **Resource management**
  - Fargate, Kubernetes

- **Execution environment**
  - Disaggregated storage, many virtualization options

- **Dataflow/analytics**
  - EMR, Athena

- **Application**
  - Neptune (graph), SageMaker (ML), Kafka (streaming)
Azure

- Good integration with Microsoft products
  - Customers that are already using Microsoft products (e.g., having existing licenses)

- Many instance types and service types

- Moved from Windows to Linux
Microsoft

- Storage
  - Azure storage (erasure coding), Project Silica
- Database
  - SQL Server
- Network
  - RDMA, FaRM
- Hardware
  - x86, FPGA (Catapult)

- Resource management
- Some research in using ML
- Execution environment
- Disaggregated storage
- Dataflow
- Dyrad, DyradLINQ
- Application
  - Project Adam (ML)
Google Cloud Platform (GCP)

- Latest among the three to come in play and smallest market share, but with good growth
- Cheapest among the three
- Fewest instance types, allows customized CPU/memory sizes
  - bill based on total CPU and memory usages, not on total instance time
- Native Kubernetes support
- Good support for cross geo-regions
Google

- Storage
  - GFS, next-gen GFS, Intel Optane
- Database
  - BigTable, Spanner (Geo)
- Network
  - FatTree
- Hardware
  - Commodity + TPU
- Resource management
  - Borg, Kubernetes, Chubby
- Execution environment
  - Non-disaggregated, containerized
- Dataflow
  - MapReduce
- Application
  - TensorFlow (ML), Pregel (graph)

Reliability

Open Source
Meta (Facebook)

- Storage
  - TAO, Haystack (photo)
- Database/NoSQL
  - Presto, MySQL, Cassandra, Memcache
- Network
  - Taiji (traffic)
- Hardware
  - Commodity, some ASIC
- Resource management
- Resource control
- Disaggregated (pods)
- Dataflow
- Application
- SVE (video), PyTorch (ML)
Common (and Important) Themes

• Scalability
  • Dist sys, local sys, networking

• Reliability
  • Failure, bugs, testing

• Security
  • Across customers, internal, regulations

• Manageability
Why is designing a computer system tricky?

• The requirement is less precisely defined, complex and volatile.

• The system has many internal interfaces as it has internal structure.

• Measure of success is not very clear

Q: How is success measured for an algorithm?
Goals of the paper

Three goals Lampson focuses on

- Functionality
- Speed
- Fault-tolerance

Three parts of design tasks he identifies

- Ensuring completeness
- Choosing interfaces
- Designing implementations
<table>
<thead>
<tr>
<th>Why?</th>
<th>Functionality</th>
<th>Speed</th>
<th>Fault-tolerance</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Does it work?</td>
<td>Is it fast enough?</td>
<td>Does it keep working?</td>
</tr>
<tr>
<td><strong>Where?</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Completeness</strong></td>
<td>Separate normal and worst case</td>
<td>Shed load</td>
<td>End-to-end</td>
</tr>
<tr>
<td></td>
<td></td>
<td>End-to-end</td>
<td>Safety first</td>
</tr>
<tr>
<td><strong>Interface</strong></td>
<td>Do one thing well:</td>
<td>Make it fast</td>
<td>End-to-end</td>
</tr>
<tr>
<td></td>
<td>Don’t generalize</td>
<td>Split resources</td>
<td>Log updates</td>
</tr>
<tr>
<td></td>
<td>Get it right</td>
<td>Static analysis</td>
<td>Make actions atomic</td>
</tr>
<tr>
<td></td>
<td>Don’t hide power</td>
<td>Dynamic translation</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Use procedure arguments</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Leave it to the client</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Keep basic interfaces stable</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Keep a place to stand</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Implementation</strong></td>
<td>Plan to throw one away</td>
<td>Cache answers</td>
<td>Make actions atomic</td>
</tr>
<tr>
<td></td>
<td>Keep secrets</td>
<td>Use hints</td>
<td>Use hints</td>
</tr>
<tr>
<td></td>
<td>Use a good idea again</td>
<td>Use brute force</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Divide and conquer</td>
<td>Compute in background</td>
<td>Batch processing</td>
</tr>
</tbody>
</table>

Figure 1: Summary of the slogans
Keep it Simple

• Do one thing at a time and do it well. Do not generalize.

• Let user make complicated decisions and put together the building blocks.

• It okay to have some complexity in corner cases that are seldom accessed.

• Get it right and keep bugs out of it.
Continuity

- Tension between desire to improve a design and need for stability
- Never break expectation of user space
- Keep a place to stand incase of interface change
Making implementations work

• Plan to throw one away

• Keep secrets of implementation

• Divide and conquer
Handling all the cases

• Normal cases must be fast

• Handle worst case gracefully

Q: If you want to evict a page in TLB, what’s the easiest way to ensure we never mess up cache of victim process?
Speed

- Split resources such that each program has its own sandbox
- Dynamic Translation
- Caching answers to expensive computations
- Use hints to speed up execution
- Compute in background
- When in doubt, use brute force

Q: Which aspect of system designing is more important than speed?
Fault-Tolerance

• End-to-end error recovery

• Log updates to record state of an object

• Transactions must be atomic or restartable
Discussion

- Can you think of any case where we have applied Lampson’s ideas throughout the course?
- Are Lampson’s ideas still applicable today?
Final Thoughts?