CS 262a: Advanced Topics in Computer Systems

Fall 2016 (MW 10:30-12:00, 306 Soda Hall)

Ion Stoica

(https://amplab.github.io/cs262a-fall2016/)

What is System Research?

Manage resources:

- » Memory, CPU, storage
- » Data (database systems)

Provide abstractions to applications:

- » File systems
- » Processes, threads
- » VM, containers
- » Naming system
- » …

This Class

Learn about systems research by

» Reading several seminal papers

» Doing it: work on an exciting project

Hopefully start next generation of impactful systems

Appreciate what is Good Research

Problem selection

Solution & research methodology

Presentation

What do you need to do?

Research oriented class project

» Groups of 2-3

One midterm exam, no final exam

Paper reading

» Submit answers to three questions for each paper before lecture

» Discuss paper during class

Research Project

Investigate new ideas and solutions in a class research project

» Define the problem

» Execute the research

» Write up and present your research

Ideally, best projects will become conference papers (e.g., SOSP, NSDI, EuroSys)

Research Project: Steps

We'll distribute a list of projects

- » You can either choose one or come up with your own

Pick your partner(s) and submit a one page proposal describing:

- » The problem you are solving
- » Your plan of attack with milestones and dates
- » Any special resources you may need

Poster session

Submit project report

Paper Reading: Key Questions

What is the problem?

What is the solution's main idea?

Why did it succeed or failed?

Does the paper (or do you) identify any fundamental/hard trade-offs?

Distributed Shared Memory

Countless papers:

» Very compelling abstraction

» Many hard challenges, so many researchers worked on it

Today

» Very few systems using shared memory, if any

» Message passing (e.g., MPI) or bulk synchronous processing (e.g., Spark) prevalent

Why did it fail?

Virtual Machine

Many papers:

- Very compelling abstraction
- Many hard challenges, so many researchers worked on it

Today

- VMs everywhere
- Containers (e.g., docker) take this concept to the next level

Why did it succeed?

What are Hard/Fundamental Tradeoffs?

Brewer's CAP conjecture: "Consistency, Availability, Partition-tolerance", you can have only 2/3 in a distributed system

Tradeoff between latency and throughput for arbitrary updates in distributed systems

» Batch request to increase throughput, but hurts latency

Grading

Project: 60%

Midterm: 15%

Class participation: 25%

Exciting times in systems research

Moore's law ending → many challenges

Many-cores machines

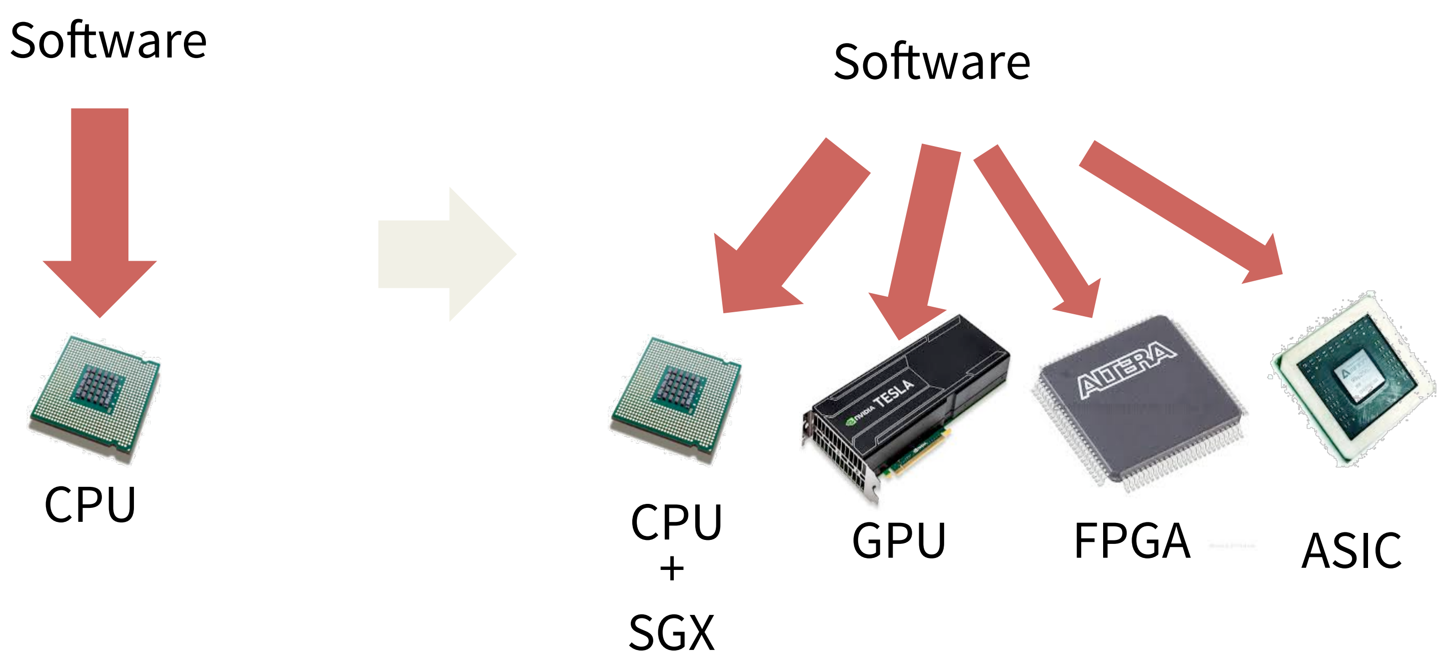
- » Amazon's X1 instances: 120 vcores and 2TB RAM

Large scale distributed systems maturing, but many challenges remain

Specialized hardware: FPGAs, GPUs, ASICs

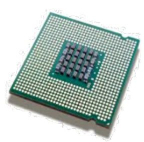
New memory technologies: 3D XPoint

Increased complexity – Computation



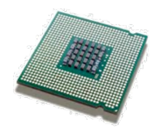
Increased complexity – Memory

2015



L1/L2 cache	~1 ns
L3 cache	~10 ns
Main memory	~100 ns / ~80 GB/s / ~100GB
NAND SSD	~100 usec / ~10 GB/s / ~1 TB
Fast HHD	~10 msec / ~100 MB/s / ~10 TB

2020



L1/L2 cache	~1 ns
L3 cache	~10 ns
HBM	~10 ns / ~1TB/s / ~10GB
Main memory	~100 ns / ~80 GB/s / ~100GB
NVM (3D Xpoint)	~1 usec / ~10GB/s / ~1TB
NAND SSD	~100 usec / ~10 GB/s / ~10 TB
Fast HHD	~10 msec / ~100 MB/s / ~100 TB

Increased complexity – more and more choices

Basic tier: A0, A1, A2, A3, A4
Optimized Compute : D1, D2, D3, D4, D11, D12, D13
D1v2, D2v2, D3v2, D11v2,…
Latest CPUs: G1, G2, G3, …
Network Optimized: A8, A9
Compute Intensive: A10, A11,…

Microsoft AZURE

t2.nano, t2.micro, t2.small
m4.large, m4.xlarge, m4.2xlarge, m4.4xlarge, m3.medium,
c4.large, c4.xlarge, c4.2xlarge, c3.large, c3.xlarge, c3.4xlarge, r3.large, r3.xlarge, r3.4xlarge, i2.2xlarge, i2.4xlarge, d2.xlarge d2.2xlarge, d2.4xlarge,…

Amazon EC2

n1-standard-1, ns1-standard-2, ns1-standard-4, ns1-standard-8, ns1-standard-16, ns1highmem-2, ns1-highmem-4, ns1-highmem-8, n1-highcpu-2, n1-highcpu-4, n1-highcpu-8, n1-highcpu-16, n1-highcpu-32, f1-micro, g1-small…

Google Cloud Engine

Increase complexity – more and more requirements

Scale

Latency

Accuracy

Cost

Security

The Unix Time-sharing System

Third major time-sharing operating system

CTSS (Compatible Time-Sharing System):
- » MIT, 1961

Multics (MULTiplexed Information and Computing System)
- » MIT, 1969

Unix stands for UNiplexed Information and Computing Systems (initially, spelled Unics)
- » AT&T, 1971

Context

Multics: 2^{nd} system syndrome (coined by Fred Brooks)

- » Following a successful system, designers become over-ambitious → complex system

"*If your project is the second system for most of your designers, then it will probably fail outright. If it doesn't fail, it will be bloated, inefficient, and icky*"

Unix a reaction to Multics

- » Uniplexed vs. Multiplexed ;-)
- » Simple, small system

“Self-Supporting System”

Use your own system, i.e., “eating your own dog food” – a lesson more valuable than ever today

Users are best developers of a system as they are in the best position to know requirements

Dogfooding origin (unverified, but nice story!):

» President of Kal Kan Pet Food would eat a can of his dog food at shareholders' meetings

Written in C

At that time all Operating Systems were written in Assembly language

- » Much easier to understand
- » Faster to develop
- » More portable (at that time there were many architectures)

33% increased in size deemed acceptable

Unix played a big role in the rapid raise of C

- » Designed by Dennis Ritchie

Minimalist design

No user-visible locks. Why?

No restrictions on number of users who can open a file, even though…

- » “contents of a file [can] become scrambled when two users write on it simultaneously”

Doesn’t enforce consistency on buffer cache

Doesn’t charge users for storage allocated to their files

Simple abstractions

Files store bytes, there is no concept of records

No distinction between “random” and sequential I/O

Files use fixed block allocation (i.e., 512B)

Simple way to implement multi-processing

- » Fork, wait, exit: trivial to share data and wait for a process (i.e., child) to terminate

Unifying Abstractions

I/O devices treated like files:

- » File and device names have same syntax and meaning
- » To a program can pass either a device or file
- » Can use same protection mechanisms like regular files

Directories special files, except

- » System control the content of directory

Unifying Abstractions (cont'd)

Pipes: unified with files

- Can easily compose simple commands to provide complex functionality
- E.g., "`grep ERROR log.txt | sort | less`"

Shell: just a program

- Reads user commands, interpret, and execute them
- Supports multitasking (backgrounding)
- Support filters, pipes

Small code base

< 50kB kernel

- A few thousands LoC
- High level language helped a lot

Only 2 man-years to write

Most successful projects start small!

Grading the paper

What is the problem?

- » Simple, powerful system that users themselves can easily evolve

What is the solution's main idea?

- » Minimalist design, unified abstractions (avoid 2^{nd} system syndrome)

Grading the paper

Why did it succeed or failed?

» Powerful, time-sharing system

» Addictive to use: interactive shell

» Open-source

» High level language made it easy to port to other architectures

Does the paper (or do you) identify any fundamental/hard trade-offs?

» Fixed block size not optimal for all apps