# Emulation of computer systems

COSC349—Cloud Computing Architecture

David Eyers

# Learning objectives

- Define terms **simulation**, **emulation** and **virtualisation**
- Understand the meaning of **host** and **guest** in the context of simulation, emulation and virtualisation
- Explain **key challenges** in software emulation of computer systems
- Describe why **cloud computing** is reliant on an ability to emulate (or virtualise) hardware in software

# Technical prerequisites for cloud computing

- Cloud computing has had extremely rapid growth
  - Many different forces have aligned
  - Not much time is spent looking backwards

- However many of its fundamental technologies have been available for far longer than the public cloud
  - Virtualisation is key underlying technology
  - … but we first talk about emulation

# Some key terms to contrast

- **Simulation**
  - Running a model of some system to observe its behaviour
- **Emulation**
  - Originally described hardware-assisted simulation
  - Now used to mean a machine imitating another machine
- **Virtualisation**
  - Adding a supervisory layer to an existing system

- These terms have shifted in their use, over time

# Key cloud requirement—decoupling

- NIST: "resources requested come from a **shared pool**."
  - Existing server software infrastructure expects to run on **particular operating systems** and hardware
  - How do you run software systems like that?

- Need a mechanism to **decouple** OSs from hardware
  - ... but computers should be deterministic machines
  - ... and software can carry out work of deterministic machines
  - therefore we should be able to pretend to provide the hardware, in software

# Key point: hardware in software

- **Simulation**: we can create a software model of the computer system we want to turn into software
  - Simulation is **often not real-time**, though
  - We also want our system to be usable like the hardware was

- **Emulation**: one machine pretending to be another so that it's actually usable as a machine
  - In particular, it will need to be **interactive**!

# Pre-cloud reasons to use emulation

- … noting that emulation typically has a high cost
  - What's emulated will be less powerful than the emulation host
- Often is used for **developing embedded systems**
  - Embedded target was difficult to debug on
  - Lack of ease of access to hardware
- Now commonplace for use in **mobile development**
  - Android emulation easily supports Android Runtime (ART)
  - iOS simulator can avoid needing to emulate hardware:
    - Apple have tight control over the i(Pad)OS software ecosystem

# Emulating the 6502 microprocessor

- A simple CPU (loved by at least Andrew and me...)
  - Three 8-bit registers: A, X and Y
  - 16-bit addresses, so 64 kilobytes of addressable RAM
  - Used in many old personal computers
    - Apple ][ series; Commodore 64; etc.

- The **computer** design around a **CPU** does input/output
  - 6502-based computers memory-map I/O devices—i.e., some memory addresses are special
  - e.g., address 0xC030 on Apple ]['s toggles the speaker

# Make some noise—specifics not in the exam

- Repeatedly toggle the speaker: create square-wave
  - Below-left shows assembly code and explanation of lines
  - Below-right is the corresponding hexadecimal machine code

```
mainloop:          A named label for jumping to.
 LDX #$70          Load 0x70 into X register.
timingloop:        Another named label for jumping to.
 DEX               Decrease X register by one.
 BNE timingloop    If X register isn’t zero, jump back.
 BIT $C030         Toggle the speaker.
 JMP mainloop      Jump back to the mainloop label.
```

```
300:
 A2 70
302:
 CA
 D0 FD
 2C 30 C0
 4C 00 03
```

# A dysfunctional emulator

- C-like pseudocode shown:
  - variable to store program counter;
  - variable to store the X register …
- **Key point**: this is a program that emulates a 6502 CPU
  - it "**executes**" 6502 machine code
  - well, five opcode types, anyway …

```
int8 opcode, register_x;
int16 address, pc = 0;
while(true){
  opcode = get_next(pc++);
  if(opcode==0xA2){
    register_x = get_next(pc++);
  }else if(opcode==0xCA){
    register_x -= 1;
  }else if(opcode==0xD0){
    pc += get_next(pc++);
  }else if(opcode==0x2C){
    address = get_address(pc);
    pc += 2;
    test_memory(address);
  }else if(opcode==0x4C){
    address = get_address(pc);
    pc = address;
  }
}
```

# Challenges building emulators—timing

- The pseudocode we showed simulates the **function** of the CPU opcodes… but that's not the complete story
- Real CPUs **take time** to execute opcodes
  - In some computers this **timing is highly precise** and matters!
  - Emulating the precise timing as well as function, is challenging!

- 6502 code example clicks the speaker periodically
  - On real Apple ][ computers, a perfect square wave produced
  - On an Apple ][ emulator, the imperfections are noticeable

# Challenges building emulators—I/O

- A computer is a CPU and equipment for interacting
  - Older computers rely on CPU control of I/O devices
    - e.g., CPU may control disk drive motors—timing may be crucial
  - Newer designs more likely delegate functionality
    - e.g., DMA, separate controller chips within I/O devices
- Delegating functions: better separation of concerns
  - ... but also increases the complexity of the systems
    - e.g., everything ends up with firmware that needs bugs fixed ...

# What I/O devices do we actually need?

- Old computers were exotic in their heterogeneity
  - e.g., multiple hard disk interfaces in one machine (IDE+SCSI…)
  - Cloud benefited from PCs becoming more regular (“boring”)
- Cloud compute node is typically just:
  - CPU cores; RAM; block storage; network interface card (NIC)
  - No need to support a complex range of graphics cards
    - Don’t need graphics output at all, or can use NIC to ship graphics
- This makes the tenant’s “computer” easier to emulate

# Specific example of an emulator: MAME

- MAME—an emulation framework
  - Commonly used to preserve vintage software's functionality
  - Currently emulates over 32,000 different individual computer systems from the past 50 years
- Old arcade computers had complex designs with multiple interacting CPUs, e.g., for sound / graphics
  - MAME supports "ROM sets" that combine the code that each CPU runs, and describes how these CPUs interact with each other and the "hardware", so that a display is shown

# MAME's support of storage devices

- Storage devices in old systems may be timing-sensitive
  - MAME has some support for common types of hardware without needing to simulate chip-level timing and interactions
- MAME floppy subsystem
  - Models how data is stored on floppy disks
  - Important this is high-fidelity, since it may be used in DRM
- MAME SCSI subsystem
  - Preserve software that supports old hardware, e.g., scanners

# Specific example of an emulator: QEMU

- QEMU: open source emulation and virtualisation
  - CPU hosting is emulation rather than simulation
  - QEMU aims to run as much of the guest system's code on the actual host CPU as possible

- Nonetheless, QEMU supports multiple CPU types:
  - x86; PowerPC; Arm; …—but host computer running one type
  - For non-native CPUs, **dynamic binary translation** cross-compiles guest machine code into code the host CPU can run

# QEMU's support of the cloud ecosystem

- QEMU's software components used in VirtualBox
- QEMU defines formats of disk images—e.g., qcow2
  - These are files that represent, e.g., virtual hard disks
- QEMU implemented many devices / subsystems:
  - PIIX3 IDE for interacting with virtual devices like hard-disks
  - VGA emulator
  - Common network interface card emulation, e.g., R1000
  - ACPI support