Why coreboot is harder than you think *and* easier than you might think possible

Ron Minnich
Google

Schedule

1000-1100 Coreboot overview

1100-1130 Depth Charge

1130-1200 Coreboot on ARM

1200-1300 Lunch/discussion/questions

1300-1400 Chromebooks and coreboot

1400-?? Your turn: you will build and boot coreboot on QEMU. And, we'll show you how we build/burn chromebooks (ARM and x86)

We encourage questions

Before we start ... save these commands

log in to your laptop

```
cd

git clone http://review.coreboot.org/p/coreboot

cd

git clone git://git.seabios.org/seabios.git seabios
```

Now that you have those commands

- Please run them *now* so that we are all ready for the tutorial
- Assuming we have a network ...
- And, also, make sure you have qemu, gnubin tools (make, gcc, etc.), ncurses-dev, bison, and flex

What coreboot is for

- Coreboot does *minimal* configuration of a platform so that the resources are discoverable configurable by other software (a.k.a. "payload")
    - payload is a kernel (Linux, Plan 9, ...) or bootloader
- Note that *it is assumed that the payload will further configure the hardware*
- *Coreboot makes the platform configurable*
- Coreboot does only as much configuration as it absolutely has to, but no more

Motivation for this talk: somebody is wrong on the internet! (http://xkcd.com/386/)

- http://tinyurl.com/cog3d8d
- "I know that the Core Boot project also tries to accomplish this, but their development process is slow and their approach seems to make the boot process more complicated than it needs to be."
- The full note is just full of errors and misunderstanding
- With a very nice set of corrections by Peter Stuge in a follow on

Coreboot/LinuxBIOS over the years

- 1999: "We don't need no steenking BIOS"
  - Let Linux do it
- 2000: Linux can't do it *all*
  - OK, we will do a lot, and then hand off to Linux
  - We'll never do ACPI for security reasons
  - And we don't care about Windows
- 2005: OK, we have to do ACPI
  - So we'll do limited ACPI, but we won't run after Linux is booted
  - And, yes, we'll do Windows
- 2008: Chipsets require System Management
  - So, we now also run after OS is booted
  - And support SMM

History: why is it called "BIOS"?

- "In the beginning", ... the BIOS was what did IO for the OS
  - Basic Input Output Subsystem
- By 1999, OSes ignored the BIOS
- In 1999, hoped this trend would continue
  - Which is why we made Linux into the BIOS -> LinuxBIOS
- The trend in the last 12 years has *reversed*
- *PCs today are more dependent on the BIOS than they have ever been*
- A sad state of affairs, probably irreversible

What a BIOS does

- 1975: "bottom half" of OS/ load top half
- 1991: load OS (e.g. 386BSD)
- 199x: configure DRAM, then load OS
- 2002: set up CPU, bug fixes, load microcode, set up DRAM, set up SMP, ...
- 2012: it's an even longer list
  - And much of it is no longer open
  - Sorry!

What we would like a BIOS to be

```
memcpy(0x100000, &bzImage, size);
((void *)(void) 0x100000)();
```

In 1991, that could have worked.

By 1999, that was impossible

What a BIOS is

	"instruction times" 1999	2012
ROM stage: getting CPU/DRAM/IO to work correctly	100	1000000000
RAM stage: set up IO, CPUs	10000	1000000
boot stage: load an OS; jump to it	100000	10000000

Really? ROM code in ~100 instructions?

- Really ...
- in 2000, on SiS 630 mainboards, we had Disk on Chip (DoC) modules
- Had to fit primary load into 256 bytes
- That code
  - initialized CPU
  - turned on RAM
  - loaded blocks from DoC
- Those blocks were the rest of LinuxBIOS
- It used to be possible
- It no longer is

What does coreboot do in 2012?

- coreboot takes a platform from a power-on state that OSes can not handle
- to a "virtual" state that OSes can handle
  - DRAM working
  - SMP working as it did in 1999, i.e. a set of largely identical CPUs
  - IO busses configured
  - microcode loaded
  - bug handlers ready in system management mode
  - ACPI configured
  - APIC configured
- It's a long list
- The result is a "virtual 1999 SMP"

Doesn't the kernel do all that work?

- Many people think that the kernel knows how to configure a platform
- It almost did, in 1999
  - Did not quite get PCI right
- But platforms today are a virtualization of the real thing (more on that later)
- The kernel is less able to configure platforms today than it could in 1999
- Due to increasing dependence on invisible BIOS actions

Comparison of coreboot effort to OS effort (I've done both several times)

Technology change	***Compares to***
Motherboard change	OS port with new drivers
Chipset change	OS port with new CPU family
New CPU implementation (e.g. P4 to Nehalem)	OS port to new architecture

- Verified with numbers from vendors
- And yet still hard to believe

BIOS is hard because hardware is hard and getting harder

- Much harder now than it was in 2000
- Simple example: programming DRAM
  - 1990 chipset: DRAM "just worked"
    - This is how most people still think of it
  - Even in the SDRAM era (1995-2002)
    - Acer chipset,8 bits SPD, one register write
  - Modern x86, 2012, thousands of bits, thousands of register writes (so much we cache it in FLASH!)
- Much less open
  - As late as 1999, most vendors documented "how to" on public sites in painstaking detail
  - In 2012, only AMD does
  - Most DRAM "turn on code" is hidden (on ARM too)

Harder, less open hardware in 2012: Percent of FLASH that is GPL code Representative system

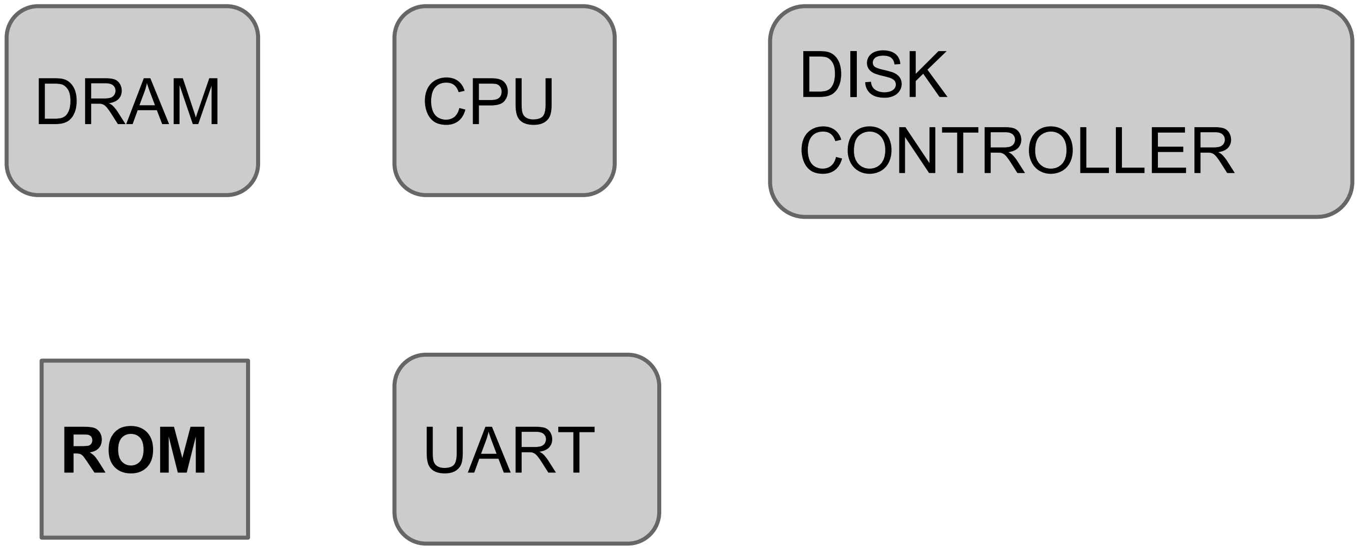
Year	Intel x86	AMD x86_64	"cellphone" ARM
1998	0	0	?
1999	100	0	?
2003	100	100	100
2005	100	100	100
2012	12	100	5

How did it get to be so hard?

- A better question: why did it used to be so easy?
- Why could 2 guys in a garage build Apple?
- The brilliance of the engineers that made that possible is often overlooked
- Ease was a result of an engineering tradeoff
- Made it easy to assemble a computer from parts
- Certain performance sacrifices were made

How hardware relates to BIOS

- 1970s/1980s world consisted of building blocks that could be dropped together



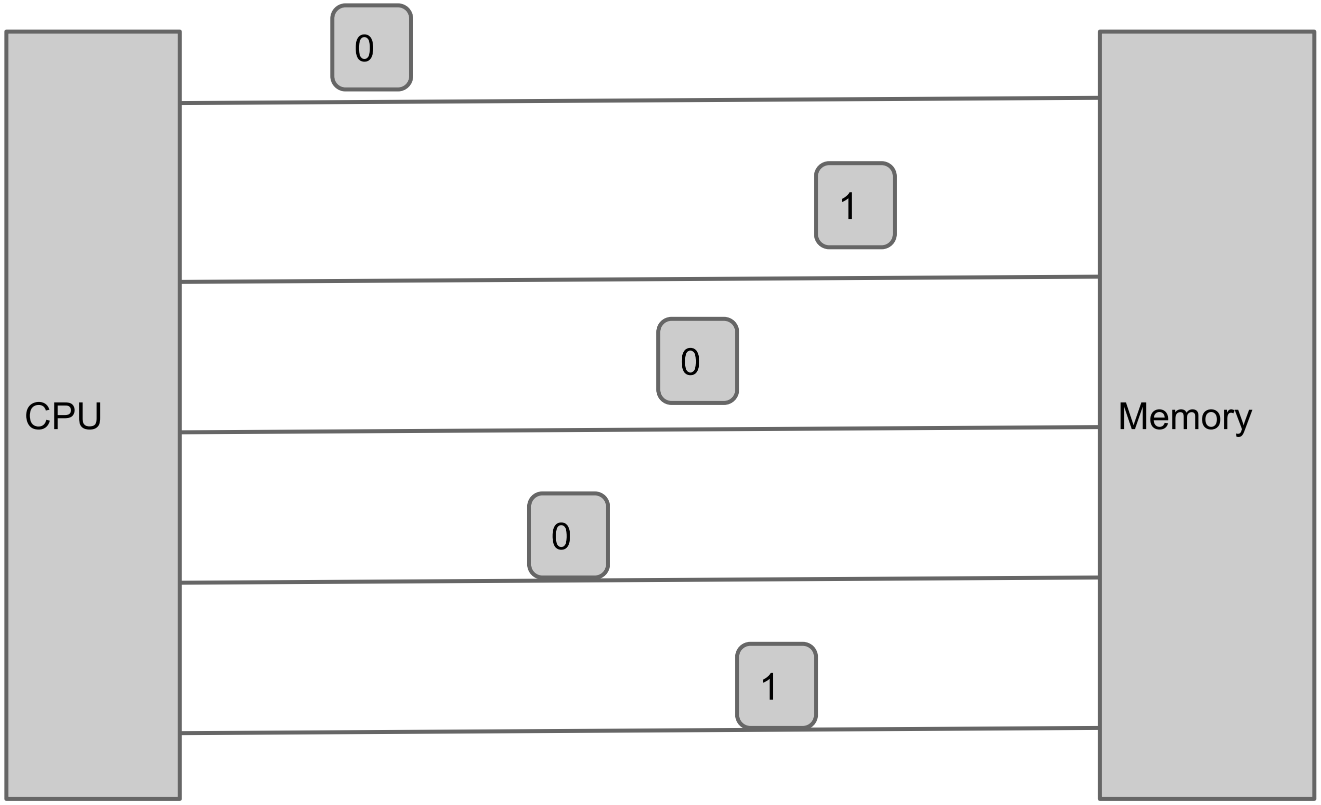
Early boot code

```
MOV $0, 0177364 //sector
MOV $1, 0177366 //block count
MOV $0, 0177362 //destination
MOV $1, 0177360 //Go!
1: BIT   $1, 0177360  // Done?
   BEQ 1b
   JMP $0 // run
```

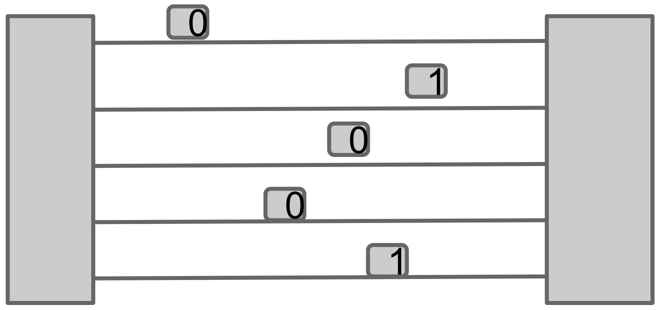
In a sense, microprocessor world presented a "virtual machine"

- The hardware was extremely tolerant
- Hardware had to guarantee that data on parallel busses had high integrity
- And skew would not be an issue
- "Skew?"

The dirty little secret: skew



Skew ....

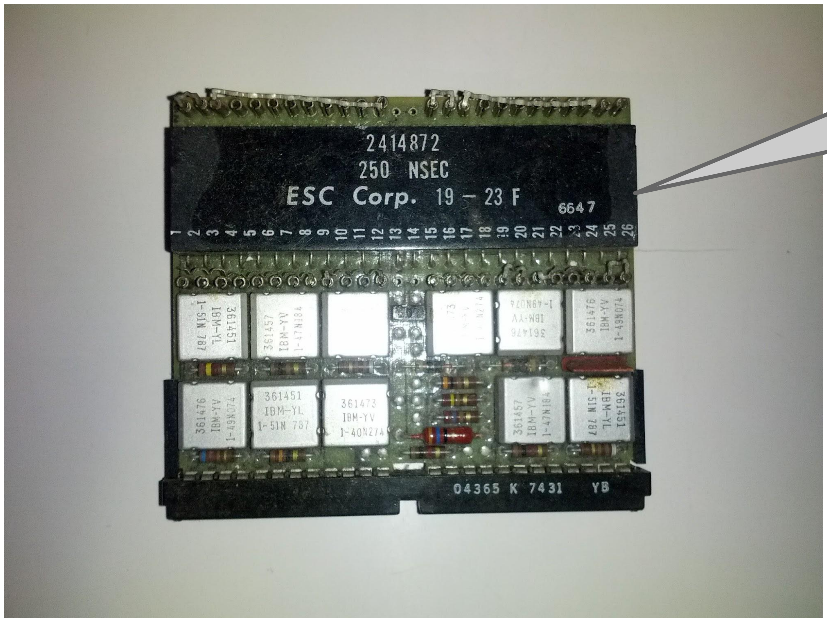


- The bits arrive at different times
- In the mini/microprocessor world, the fix was to go slow
    - i.e. spec the bus timing so skew was not seen
- And use 'asynchronous' busses
- In the mainframe/supercomputer world, the fix was to (by hand) tune per-wire delay lines
- And use synchronous busses
- So they went faster:
- 1991 Cray: 12,000 MiB/sec
- 1991 microprocessor: 32 MiB/sec

Improving performance: 1995 (or so) synchronous DRAM

- DRAM, CPU now have a common clock
- Timing parameters located on DRAM modules in serial ROM (SPD)
  - Contains timing parameters for DRAM chips
  - BIOS knows timing for CPU chipset
- Compute intersection and program chipset
- Problems
  - DRAM modules not always accurate
  - Boards have issues
  - Chipsets do not always work
- BIOS had to "know" the "quirks"
- Needed a better way to tune delay

Adjusting skew in a System/370



Delay line selectable in 250 ns increments

How you select a delay: jumpers
Many of these delay lines would be in a typical system

2002: training comes to DRAM bus

- New (to most of us) technology: training
- "Training" is the process of iteratively putting data on a bus, observing bus behavior, and tuning *per-wire* on-chip delay lines to optimize performance and minimize error
- It's quite hard to get right
  - In some cases it has taken two years for a new chipset -- by the vendor
- In the last 10 years it has gotten very complex
  - Takes up to 600 milliseconds on modern chipsets

2010: training comes to *every* bus

- Busses that talk to PCI controllers
- Busses that talk to the display
- Simple almost-serial busses that talk to simple devices
- Almost every bus has to be trained
- And training can require a side-band interaction with the device to get parameters
    - video devices talk over DDC link to manage training
- Much training happens after memory is up
- More time taken before the OS can run

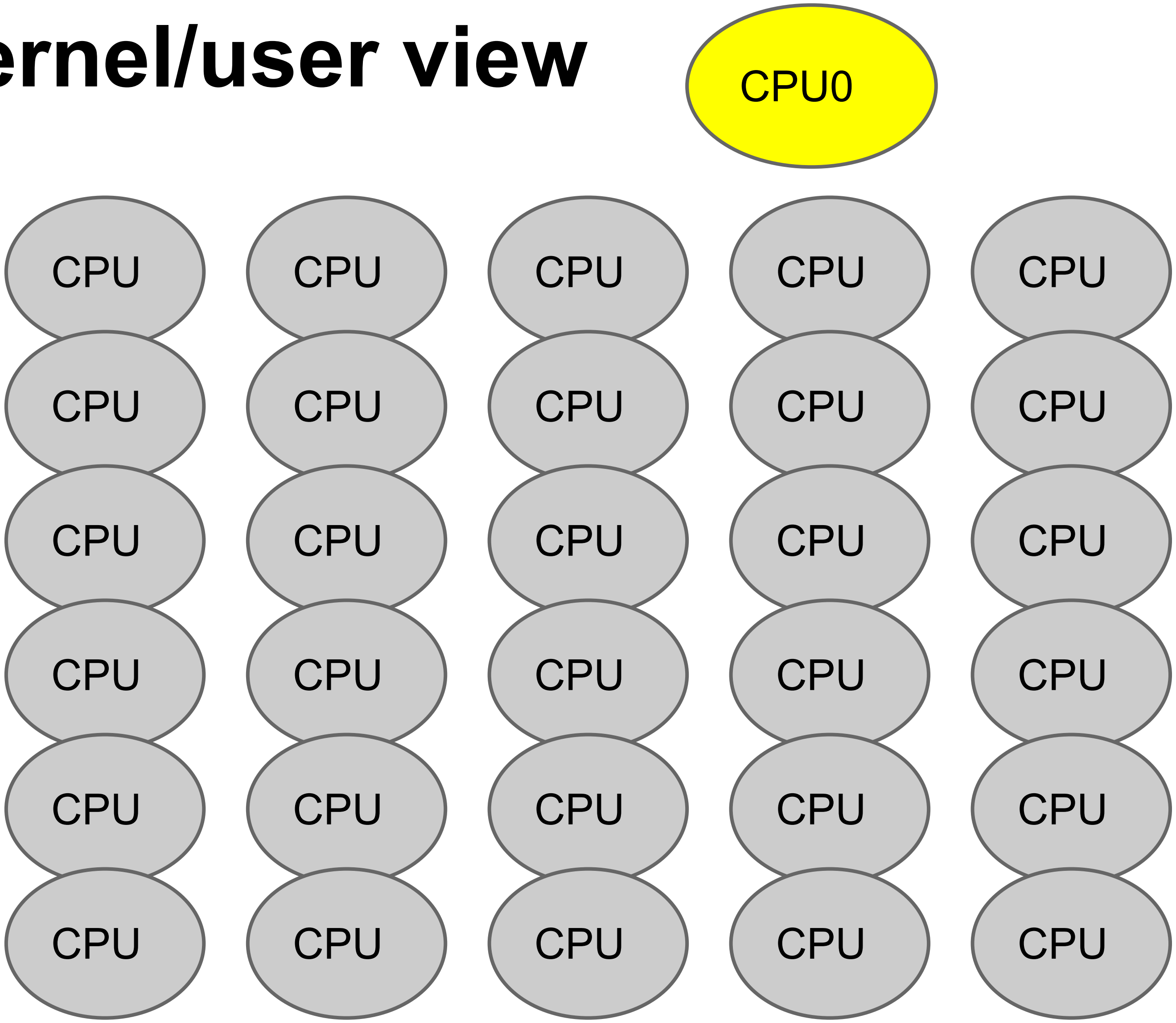
Training

- All of it visible at the BIOS level
- Almost none of it visible at the OS level
- In other words, the task of the BIOS is to configure complex, messy hardware into a simple form that the OS can understand
- Which fools OS people into thinking they know more about hardware than they do
- Hence those error-filled notes on the mailing lists about how simple BIOS can be
- The OS people no longer see the real hardware!

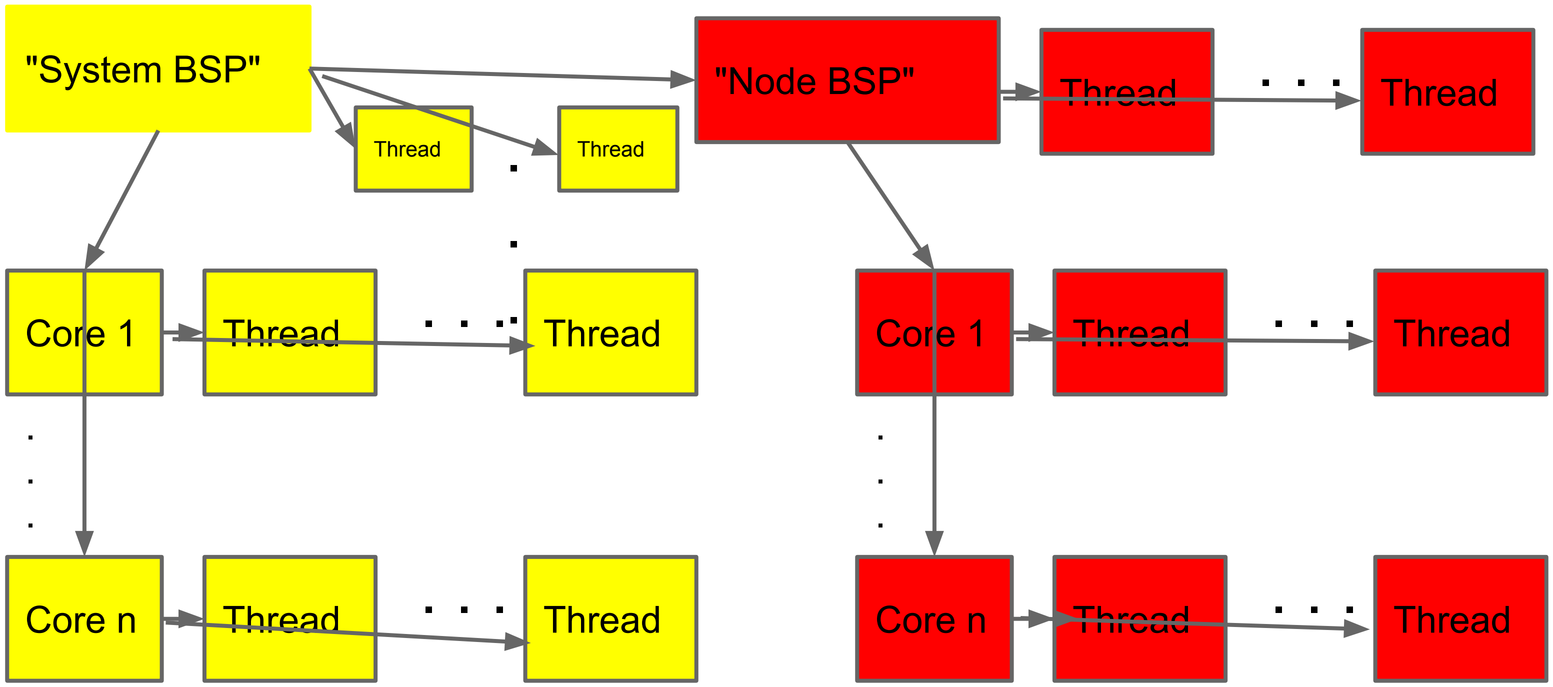
Virtual hardware example: SMP startup

- Goal: provide users with a "sea of CPUs"
- e.g., cat /proc/cpuinfo does not show different *types* of CPUs
- You don't think about it
- For the most part, kernel does not either
- But at the BIOS level, it is very visible!

The kernel/user view



The BIOS view



- Hardware is barely working at boot
- Have to at least load microcode on most modern systems
  - But only on some of the cores ...

Recommended SMP bringup

- Is NDA for new systems. Sorry!
- What the kernel does
  - send a broadcast SIPI to wake the cores up
  - Cores come up and self-configure
  - Quite elegant
  - Only works because BIOS did the hard part
- What coreboot does
  - carefully send one core at a time an SIPI
  - Tells core its starting IP
  - core loads SP from a global variable
  - Core zeros variable
  - which lets coreboot know it can wake the next core
- Why the difference?

Why the difference?

- At startup, different cores have very different roles/capabilities
- One core turns on memory for all sockets
- Some cores do socket setup
- Some cores turn on threads
- Some threads do per-thread setup
- Lots of texture
- Goal of BIOS is to hide this texture and attendant complexity as much as possible

Kernel SMP bringup

- Everyone wakes up at same time --> maximum parallelism
- Obvious question: can we use kernel algorithm in coreboot?
- The old answer: yes
  - on a set of P4 xeons ....
- The new answer: no
  - on newer systems, the cores are heterogeneous
- We actually tried the new system
- It worked on *almost* everything
- Except newer cpus!

The lesson

- In 2000, we created SMP startup for kernel
  - First GPL'ed BIOS-level SMP startup
  - "Let Linux do it"
- Worked on Intel
- Did not work due to K7 issue
  - On K7 all cores start up
  - BSP selection in software
  - BIOS therefore has to do SMP startup
- Moved it directly into coreboot
- But a kernel-level SMP startup will no longer work in the BIOS
- Much more complex at BIOS level

BIOS sets up CPUs so simple SMP models work

- The BIOS sees all the ugly SMP startup
    - And hides it
    - So the simple kernel-level SMP startup works
- Extend that idea to the rest of the hardware
- And now you know what a BIOS does
- And it's certainly not what it used to do ...

So it's hard. But not impossible.

- I hope I did not scare too many people off
- The goal of coreboot is to make a very hard problem less hard
- And to make it open source
- To do so we structure coreboot in a way designed to make adding new boards easy
- When a new board is needed, very little code changes

Example for a modern board

- Consider the case of the Samsung Chromebook and Chromebox
- One is a laptop, the other a desktop
- Laptop has a screen and keyboard built in
- Desktop supports up to two displays
- The laptop has an embedded controller (EC)
- Laptop can support WiFI and 3G
- Those are very different systems
- So let's take a look, first at the tools, then the code

Coreboot uses git, gerrit, and jenkins

- Git you know well
- Gerrit is the code base management tool developed for android (demo)
- Jenkins is a "continuous build" framework
- When a CL is received, Jenkins starts off a build of *every single supported board* to make sure nothing breaks
    - One company told us how hard it was to manage 27 boards -- for just their own hardware!
    - we manage 10 times that many, for many vendors/chipsets, and most of it is automated
- Jenkins will block CLs that break the build

Coreboot uses kconfig for configuration

- We went through several config tool iterations
- It was clear that Linux kconfig was the right tool for many reasons
- Made the change in 2009
- Demo

Coreboot has a wealth of utilities

- When creating a new mainboard, there are things you have to learn
- As the architectures and chipsets have gotten more complex, we need more such tools
- (demo)

coreboot has support for dynamic resource discovery and allocation

- Perhaps our strongest capability
- Systems such as ARM tend to be very static
  - Simple config file can define the entire system
- PCs are very dynamic
  - DRAM, devices, CPU types, and so on
  - PCI enumeration can be very complex, especially with bridges
- Coreboot is designed such that one can specify classes of resources
- Coreboot can manage very complex systems without requiring complex build-time spec

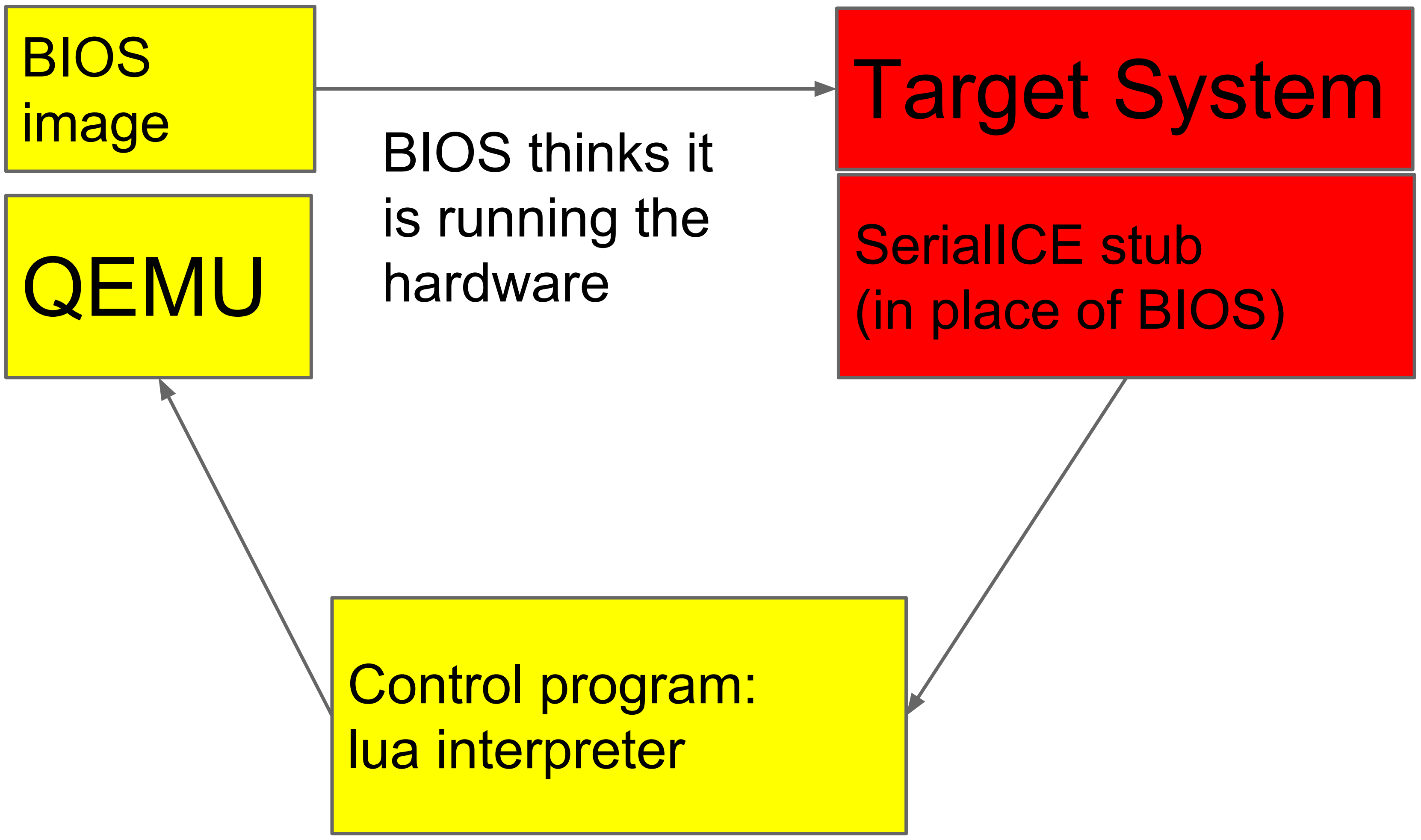
Coreboot supports powerful debug tools

- Full user-mode emulation environment
- Runs under qemu for learning
- gdb stub
- SerialICE: a full in-circuit-emulation environment without the cost
- Possibly the neatest coreboot tool

SerialICE

- The problem:
- Need to run a test BIOS on a system without working memory or most IO
- gdb stub can not function in that world: no memory!
- Serialice consists of a very tiny "stub" that is flashed onto the target board
- Stub supports simple remote operation command set
- Test BIOS actually runs under qemu *on the host*

SerialICE: run BIOS on host, have it control target


BIOS image
QEMU
BIOS thinks it is running the hardware
Target System
SerialICE stub (in place of BIOS)
Control program: lua interpreter

SerialICE

- As the BIOS runs on QEMU, it will perform a set of IO and memory ops
- These ops are transparently relayed over the SerialICE link to the stub running on hardware
- Can completely recreated IOs used to bring a board up
- Only limit is timing-dependent operations are hard

SerialICE has let us answer really hard questions about hardware

- Extremely useful for very early stages of DRAM startup
- Many other uses in different phases of bringup
- In some cases it is the difference between success and failure
- Can be used to discover obfuscated hardware issues

And, finally, the community

- There are a lot of great people working on coreboot
- IRC and mailing list
- Some at companies (e.g. Google) others at universities
- coreboot.org
- Always happy to help
- And we're always looking for new members!

So, yes, it *is* hard

- But it is a chance to learn about the lowest levels of the hardware that few people know
- And you can build on coreboot to do some really innovative things
- In ways that are simply not possible using standard BIOSes
- Since, after all, they are stuck with a 30-year legacy compatibility burden
- We don't have that problem
- Although we *do* boot windows

Conclusion

- The system that the OS people see is an illusion
- Constructed by the BIOS
- Coreboot allows you to see what's going on underneath
- And also provides a powerful environment for customizing generic platforms
- We welcome new members
- See coreboot.org for more information

Quick walkthrough of a real example

We're going to build coreboot and seabios and boot a kernel

You need qemu; hope you have it.

If not, we can help you install it

I'll walk through it, then YOU will do it :-)

Setting it up

git clone http://review.coreboot.org/p/coreboot

cd coreboot

make menuconfig\to the payloads menu and set it up for

no payload

```
Architecture (x86)  --->
        Chipset  --->
      Generic Drivers  --->
        Console  --->
        System tables  --->
        Payload  --->
        VGA BIOS  --->
        Display  --->
```

```
Add a payload (SeaBIOS)  --->                              │ │
  │ │       SeaBIOS version (stable)  --->
│ │
  │ │    [*] Use LZMA compression for payloads (NEW)
```

Change to

```
(X) None                                          | |
    | |            ( ) An ELF executable payload              |
|            ( ) SeaBIOS                                 | |
    | |            ( ) FILO                                    | |

   | |
```

Make coreboot

```
make
```

Creates a rom image that has an embedded filesystem called 'cbfs'

To examine contents,

```
./build/cbfstool build/coreboot.rom print
```

This won't boot: no payload

```
Name                              Offset     Type         Size
cmos_layout.bin                   0x0        cmos layout  1160
fallback/romstage                 0x4c0      stage        9817
fallback/coreboot_ram             0x2b80     stage        
30274
config                            0xa200     raw          2357
(empty)                           0xab80     null         
217320
```

Get seabios as a payload and make

git clone git://git.seabios.org/seabios.git seabios

etc.

make

```
  Working around non-functional -combine
  Build default config
#
# configuration written to /root/seabios/.config
#
  Working around non-functional -combine
  Build Kconfig config file
  Compiling IASL out/acpi-dsdt.hex
out/acpi-dsdt.dsl.i    570:            Return(0x01)
Warning  1104 -                                ^ Reserved method should not return a
value (_L00)

out/acpi-dsdt.dsl.i    573:            Return(\_SB.PCI0.PCNF())
Warning  1104 -  Reserved method should not return a value ^  (_E01)
```

Requires python2 ...

```
Version: rel-1.7.1-35-g02203b5-
20121015_122040-chromix
  File "./tools/layoutrom.py", line 76
    print "Error: Fixed section %s has non-zero
alignment (%d)" % (
```

so

make PYTHON=python2

…

Add the seabios payload to coreboot

cd ~/coreboot

./build/cbfstool build/coreboot.rom add-payload -f ~/seabios/out/bios.bin.elf -n fallback/payload

```
Name                              Offset     Type          Size
cmos_layout.bin                   0x0        cmos layout  1160
fallback/romstage                 0x4c0      stage         9817
fallback/coreboot_ram             0x2b80     stage
30274
config                            0xa200     raw           2357
fallback/payload                  0xab80     payload
123960
(empty)                           0x29000    null
93288
```

Run it under qemu

```
qemu-system-x86_64 -m 256  -serial stdio -b
ios build/coreboot.rom -cdrom ~/Core-
current.iso -boot d
```

Now it's your turn ... after the rest of the talks! (and we'll see this again)