The Trouble with Hardware

Timothy Roscoe

Systems Group, ETH Zurich

November 30th 2017

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Many thanks!

And all the ETH Systems Group!

The Gap.

For many commercially relevant workloads, cores spend much of their time in the OS.

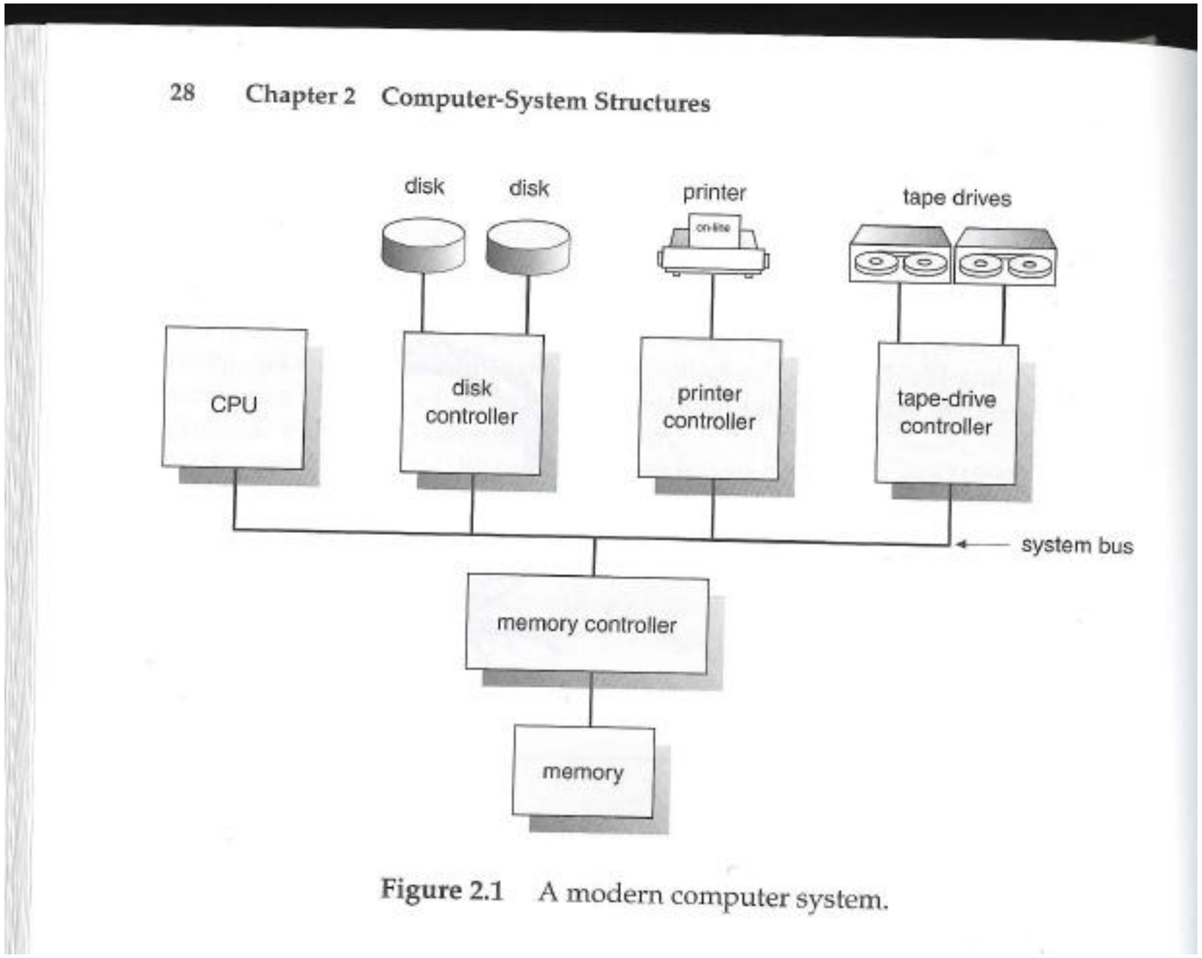
BUT:

- Processor architects ignore OS designers
  - Simply don't understand the OS problem
  - Cores rarely evaluated with >1 app running anyway
- HPC people try to remove the OS
  - And then blow the rest of their s/w development budget putting it back in a user library.
- and OS design people?
  - Complain among themselves and try and deal with it
  - Don't even try to influence hardware

w/ Andrew Baumann, Livio Soares, Jeff Mogul

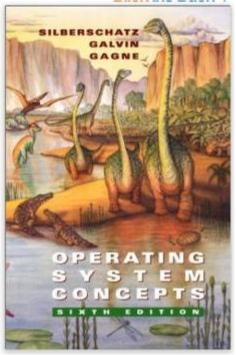
SO, WHAT IS THE TROUBLE WITH HARDWARE?

Lies we teach our children



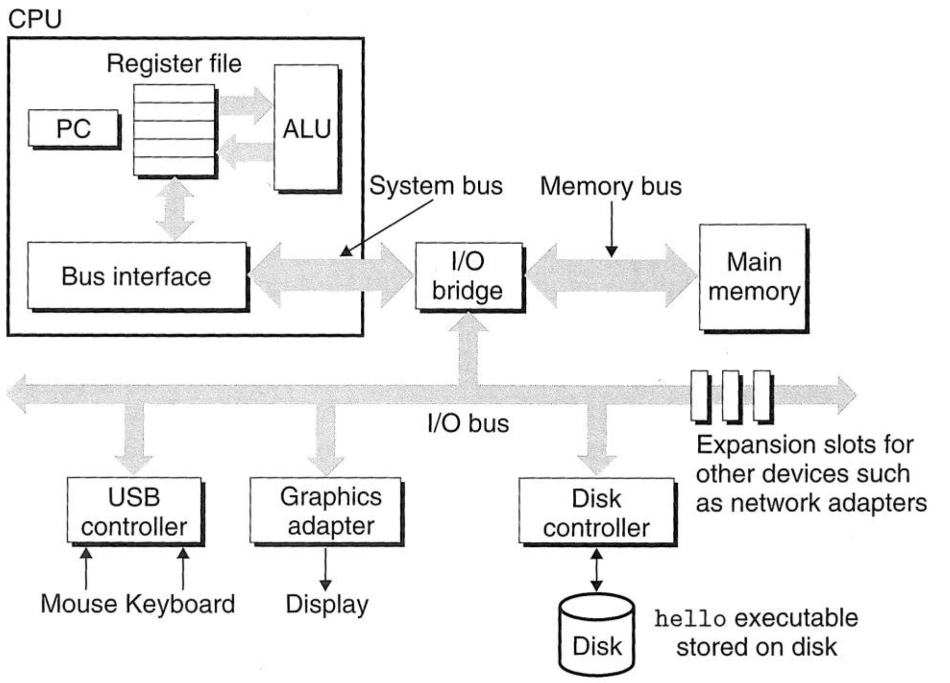
28 Chapter 2 Computer-System Structures

Figure 2.1 A modern computer system.



Lies we tell our children

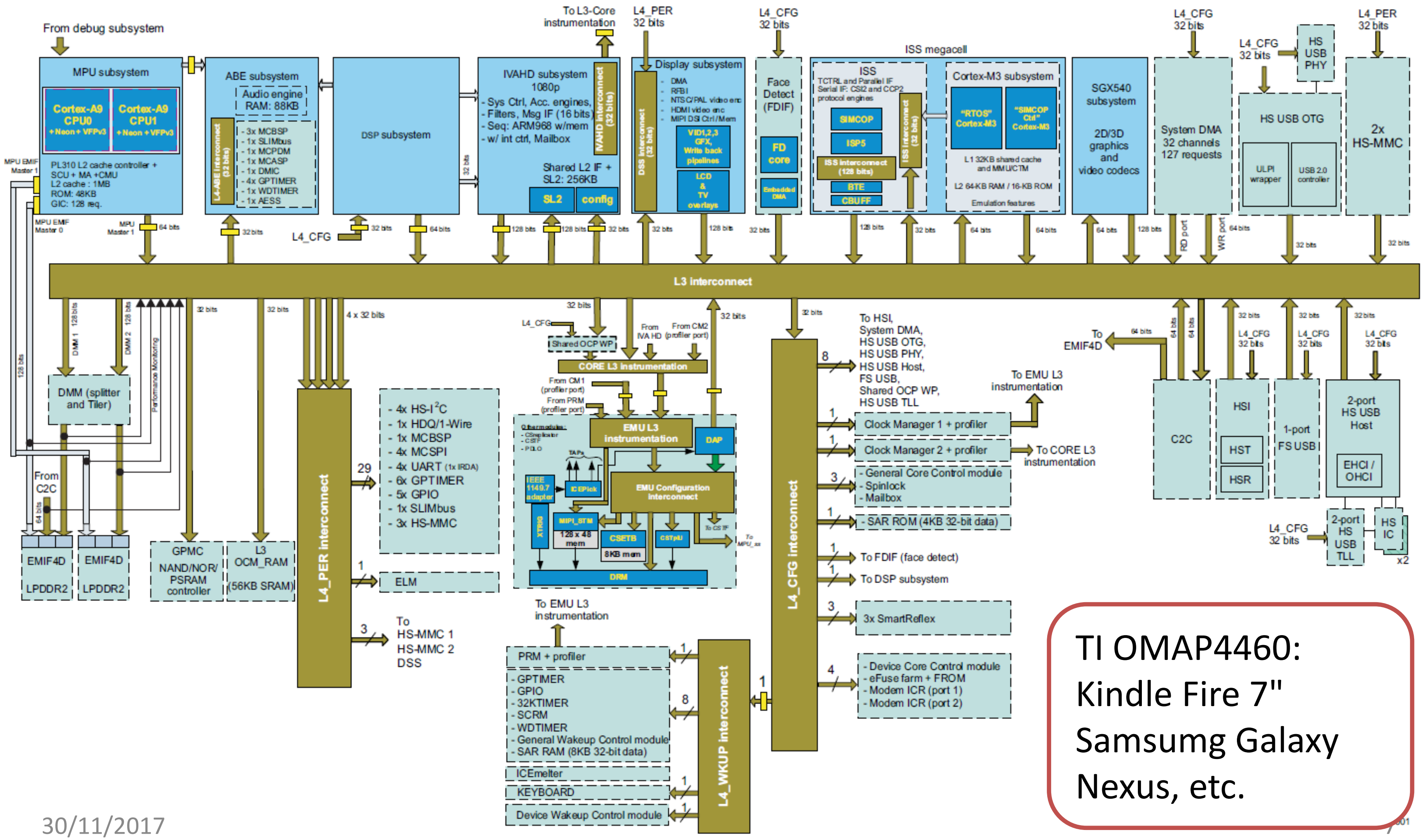
Figure 1.4
Hardware organization of a typical system. CPU: Central Processing Unit, ALU: Arithmetic/Logic Unit, PC: Program counter, USB: Universal Serial Bus.



systems, but all systems have a similar look and feel. Don't worry about the complexity of this figure just now. We will get to its various details in stages throughout the course of the book.

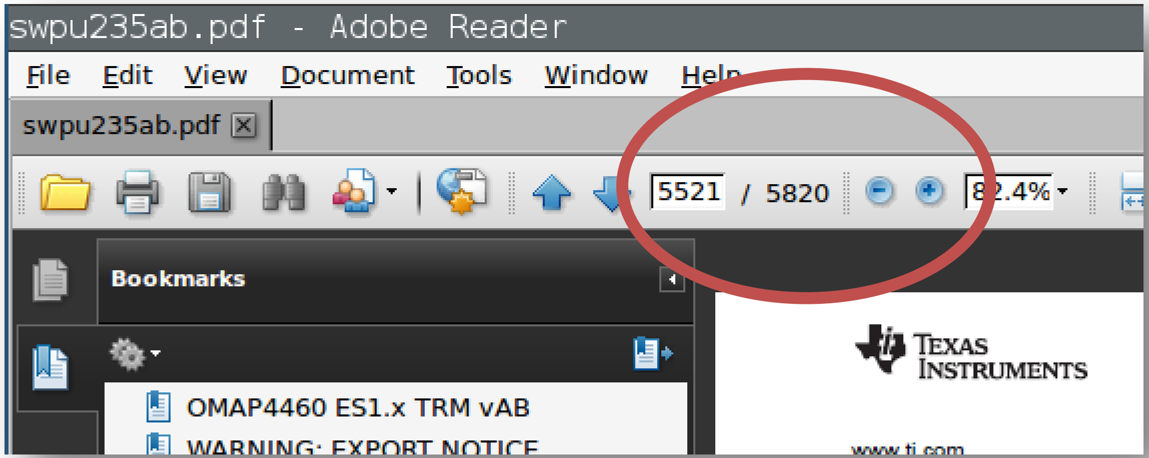
Computer Systems, A Programmer's Perspective, Bryant & O'Hallaron, 2011

A great way to frighten students

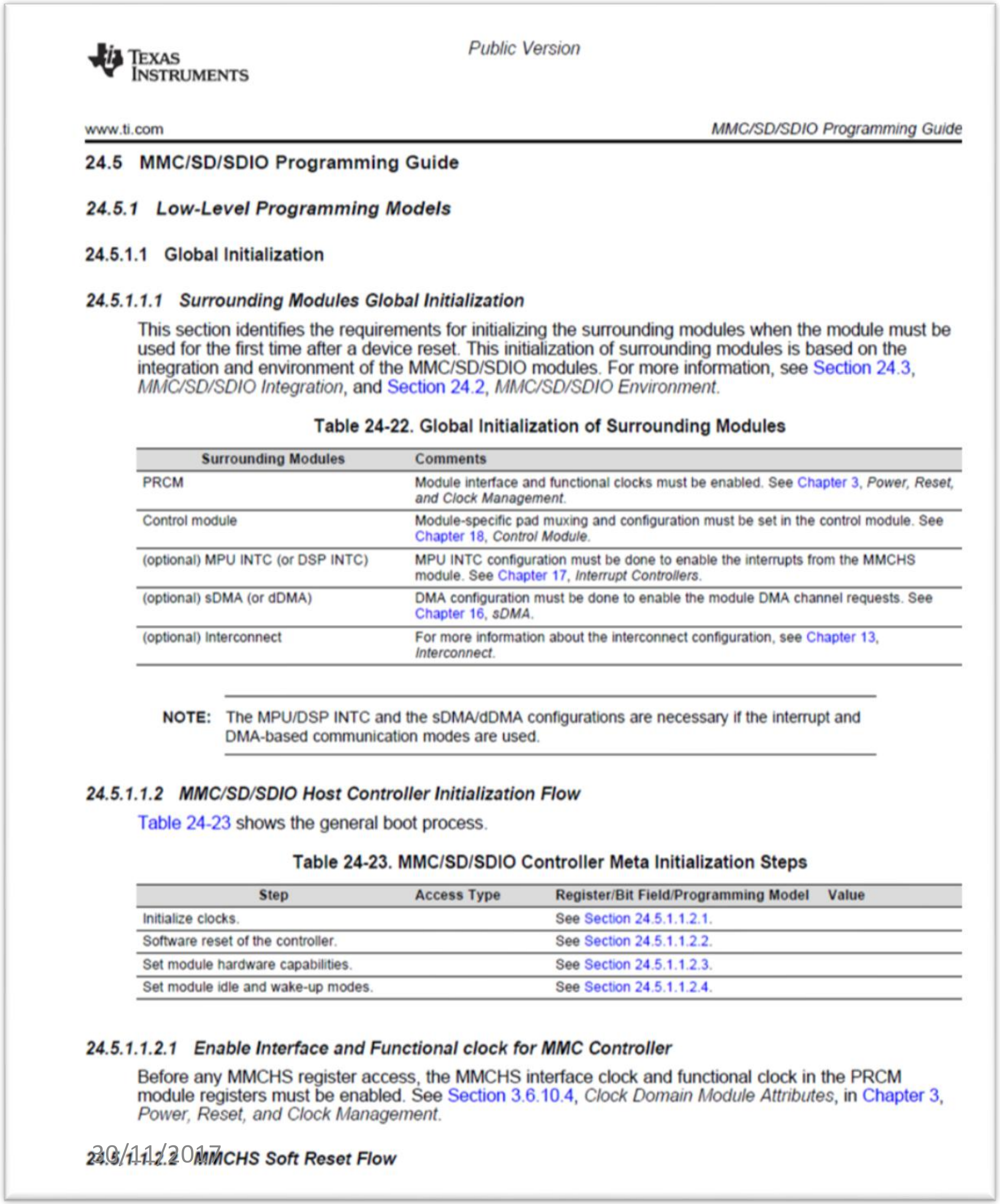


TI OMAP4460:
Kindle Fire 7"
Samsumg Galaxy
Nexus, etc.

A great way to frighten students



- pp. 3-63: Table of contents
- pp. 64-88: List of figures
- pp. 89-258: List of tables



Public Version

www.ti.com MMC/SD/SDIO Programming Guide

24.5 MMC/SD/SDIO Programming Guide

24.5.1 Low-Level Programming Models

24.5.1.1 Global Initialization

24.5.1.1.1 Surrounding Modules Global Initialization

This section identifies the requirements for initializing the surrounding modules when the module must be used for the first time after a device reset. This initialization of surrounding modules is based on the integration and environment of the MMC/SD/SDIO modules. For more information, see Section 24.3, *MMC/SD/SDIO Integration*, and Section 24.2, *MMC/SD/SDIO Environment*.

Table 24-22. Global Initialization of Surrounding Modules

Surrounding Modules	Comments
PRCM	Module interface and functional clocks must be enabled. See Chapter 3, *Power, Reset, and Clock Management*.
Control module	Module-specific pad muxing and configuration must be set in the control module. See Chapter 18, *Control Module*.
(optional) MPU INTC (or DSP INTC)	MPU INTC configuration must be done to enable the interrupts from the MMCHS module. See Chapter 17, *Interrupt Controllers*.
(optional) sDMA (or dDMA)	DMA configuration must be done to enable the module DMA channel requests. See Chapter 16, *sDMA*.
(optional) Interconnect	For more information about the interconnect configuration, see Chapter 13, *Interconnect*.

NOTE: The MPU/DSP INTC and the sDMA/dDMA configurations are necessary if the interrupt and DMA-based communication modes are used.

24.5.1.1.2 MMC/SD/SDIO Host Controller Initialization Flow

Table 24-23 shows the general boot process.

Table 24-23. MMC/SD/SDIO Controller Meta Initialization Steps

Step	Access Type	Register/Bit Field/Programming Model	Value
Initialize clocks.		See Section 24.5.1.1.2.1.	
Software reset of the controller.		See Section 24.5.1.1.2.2.	
Set module hardware capabilities.		See Section 24.5.1.1.2.3.	
Set module idle and wake-up modes.		See Section 24.5.1.1.2.4.	

24.5.1.1.2.1 Enable Interface and Functional clock for MMC Controller

Before any MMCHS register access, the MMCHS interface clock and functional clock in the PRCM module registers must be enabled. See Section 3.6.10.4, *Clock Domain Module Attributes*, in Chapter 3, *Power, Reset, and Clock Management*.

24.5.1.1.2.2 MMCHS Soft Reset Flow

Initializing the SD reader

(page 5503 ff.)

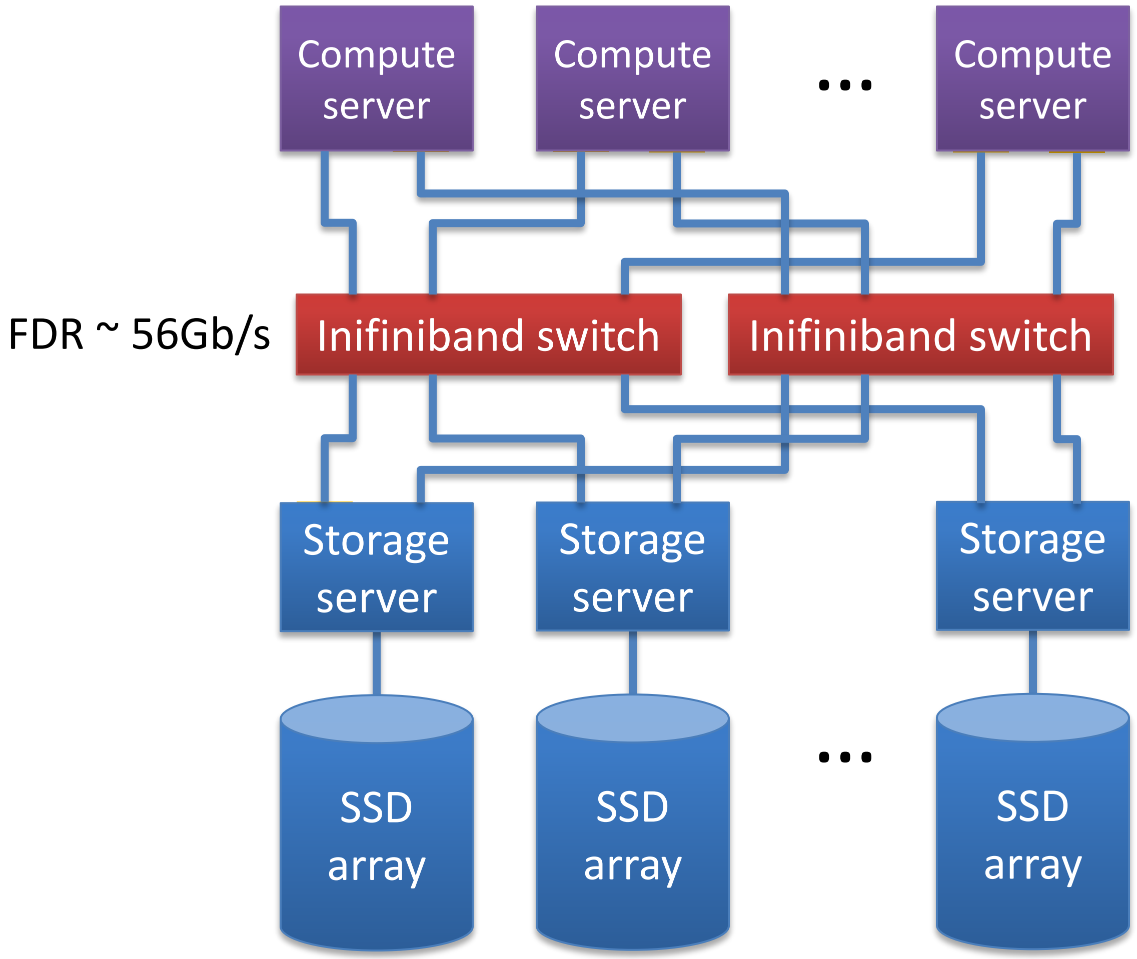
and so on for 7 pages.

Is this a computer?

Still programmed *mostly* as a classical distributed system

Typical rack-scale architecture

(in **research**, at least)

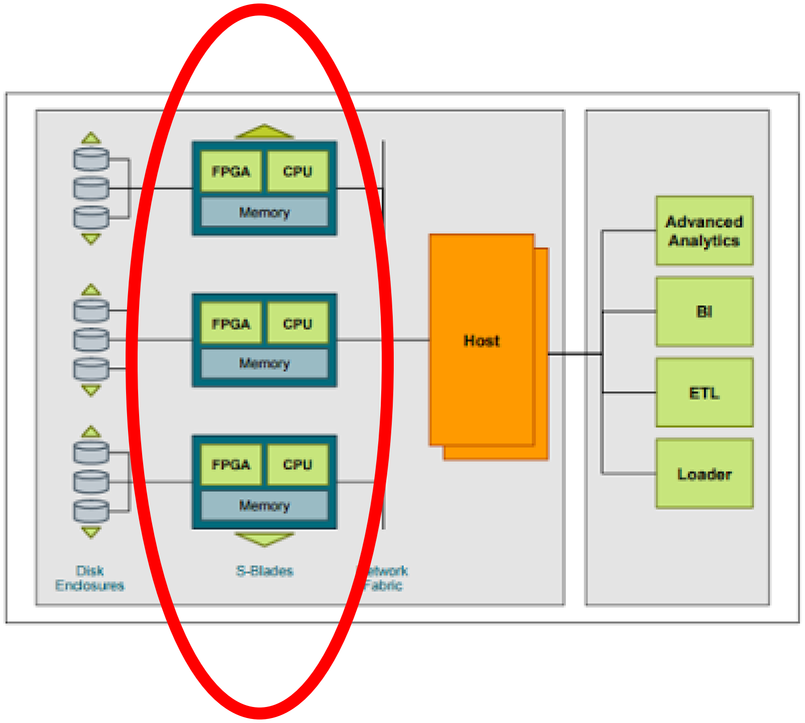


Q. Message latency in this network?

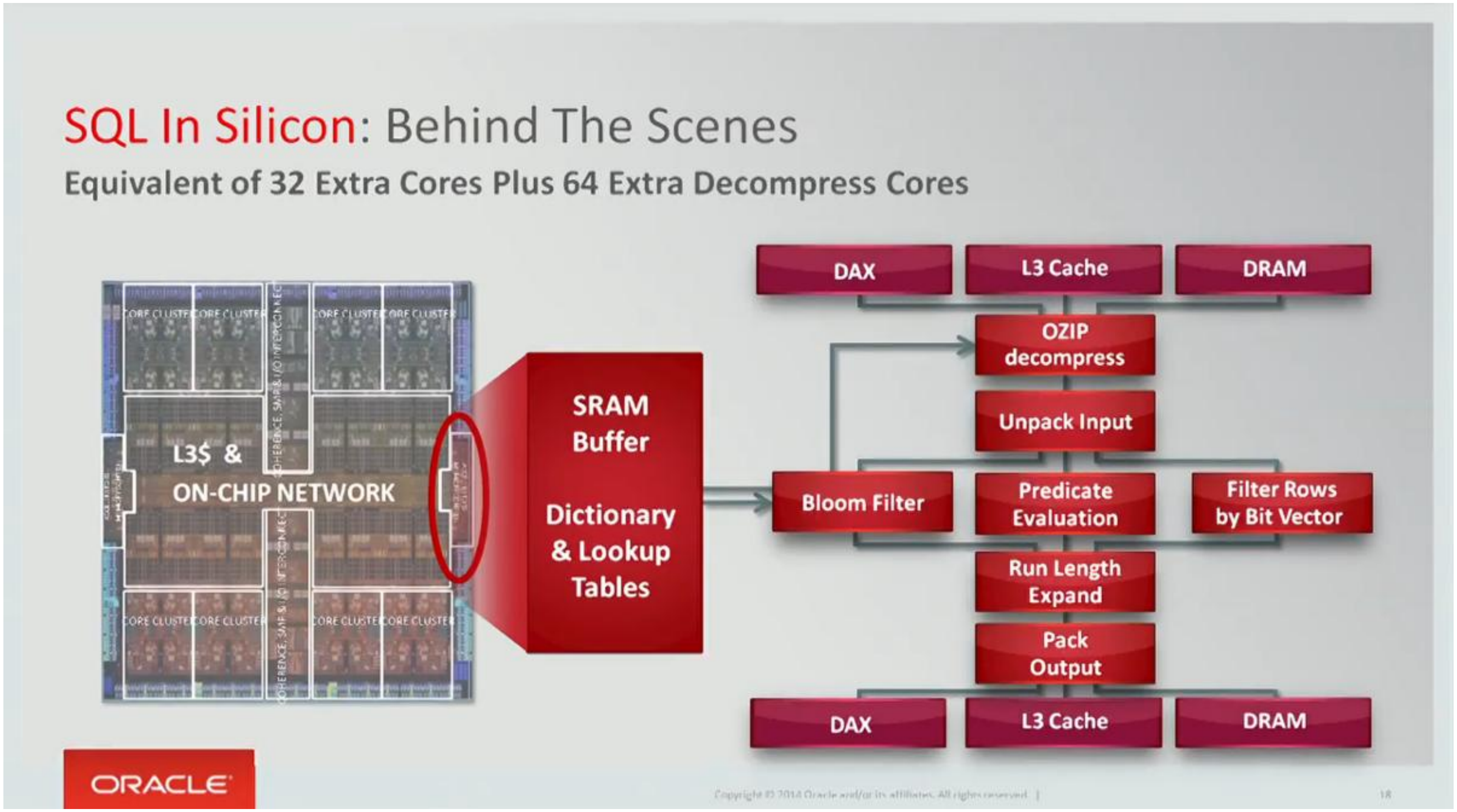
A. ~ 0.7μs, or ~10 LLC misses.

⇒ need to think of this as ***one big machine***

But it's not typical.



But it’s not typical.



SQL In Silicon: Behind The Scenes

Equivalent of 32 Extra Cores Plus 64 Extra Decompress Cores

Copyright © 2014 Oracle and/or its affiliates. All rights reserved. | 18

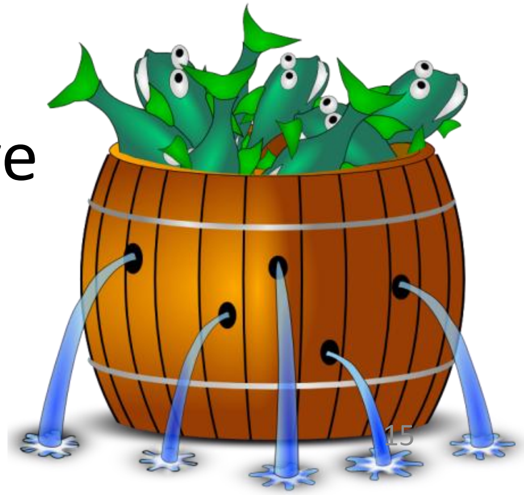
So, hardware is too complicated.

What clever techniques do OS developers use to mitigate this problem?

1. The powerful high-level abstractions of **C**
2. Kernel **modules**
3. Server processes and **daemons**
4. Er…
5. That’s it.

The Barrelfish research OS

- Written from scratch, 2008-present
  - Industry help: Microsoft, HPE, Huawei, Oracle Cisco, VMware, Xilinx, ARM, Cavium, Intel, …
- Used for research and teaching
- Currently ~ 1m lines of code
  - MIT open source licence
  - ARMv7-A, ARMv8, x86_64, KNL
  - Previously: ARMv5, ia32, SCC, Beehive
  - See www.barrelfish.org



WHAT DID WE LEARN FROM BARRELFISH?

Learnings

We started trying to solve three challenges:

- Scaling to large core counts
- Dynamic, heterogeneous cores
- Complex memory hierarchies

We ended up identifying two big problems:

- Sheer complexity of hardware for software
- Ossification of hardware/software ecosystem

Hardware is changing faster than system software

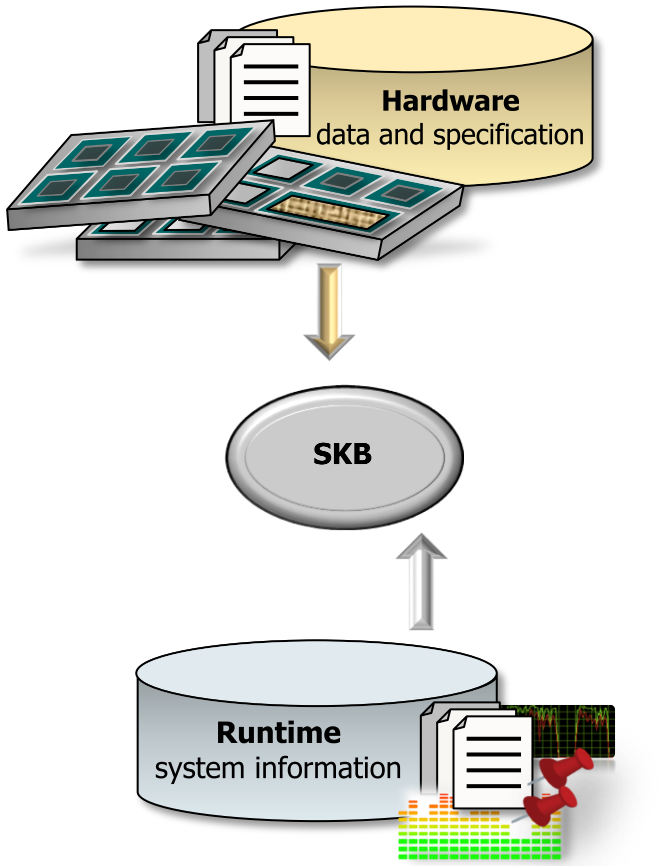
- CAD systems make it easy to cut’n’paste
- High volumes for SoCs lead to diversity
- End of Dennard scaling $\Rightarrow$ specialism

System software is getting harder to change

- The OS retreats from most of the hardware
  - Heterogeneous cores?
  - Non-cache-coherent memory?
  - etc.
- OS can’t adapt to changing tradeoffs
  - “Least common denominator” tuning
- Today Linux is ***at best*** a small component of *“that which manages the machine”*

1ST TRY: LET’S REPRESENT THE MACHINE IN PROLOG

SKB – System Knowledge Base



- Basic OS service
  - Boots early
- Holds:
  - Hardware info
  - Runtime state
- Queried by:
  - OS services
  - Applications
- Rich semantic data model

Plenty of design options

- Knowledge-representation frameworks
- Database
- RDF
- Logic Programming, inference
- Description Logics
- Satisfiability Modulo Theories
- Constraint Satisfaction
- Optimization
- etc.

Initial choice:
ECLiPse CLP solver:
Prolog + constraint extensions
(circa 2009!)

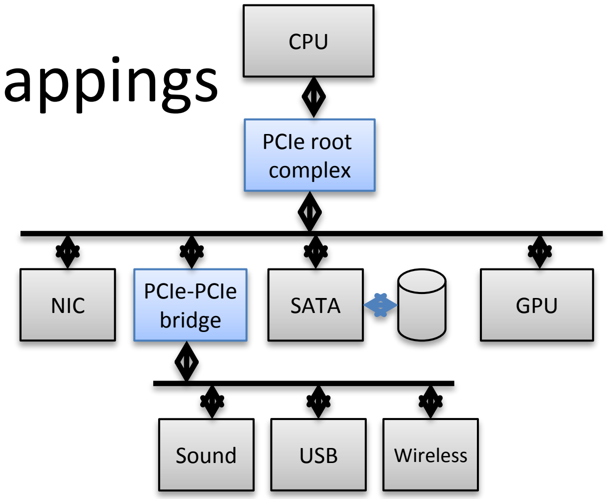
A few SKB applications

- General name server / service registry
- Coordination service / lock manager
- Device management
  - Driver startup / hotplug
- PCIe bridge configuration
  - A surprisingly hard CSAT problem!
- Intra-machine routing
  - Efficient multicast tree construction
- Cache-aware thread placement
  - Used by e.g. databases for query planning

Example 1: PCIe bridge configuration

- Hierarchical allocation of physical address ranges
  - Natural power-of-two aligned
  - Three disjoint memory types
  - "Holes" in physical address space
  - Some devices can't be moved
  - Odd constraints on some address mappings
  - "Quirks"
  - HotPlug
  - …

Adrian Schüpbach,
Simon Peter,
Andrew Baumann



How do others deal with this?

- Mostly, they don't.

- Linux uses BIOS allocation and runs a fixup procedure
  - Configures missing devices if no bridge reprogramming needed
  - Otherwise fails
- Windows Vista, Server 2008: PCI Multi-Level Rebalance
  - Can move bridges to a place with bigger free space
  - Machine may appear to freeze for a few seconds
- IBM US patent 5,778,197 (1998): "Method for allocating system resources in a hierarchical bus structure"
  - Recursive bottom-up algorithm to allocate resources
- People have published genetic algorithms for this problem (!)
- 25 years after PCI 1.0 standardization, no complete solution exists.

We coded it in constraint Prolog (CLP)

- Cleanly separate:
    1. "Ideal" allocation computation (in Prolog)
    2. Ad-hoc constraints (errata, quirks, etc.)
    3. Register read/write code (in C)
- A new quirk is 1-4 lines of portable Prolog
- CLP boots before devices
    - Runtime milliseconds vs. microseconds

In 2012, we published a TOCS paper on how to configure a 20-year old hardware standard.

Example 2: Database thread placement

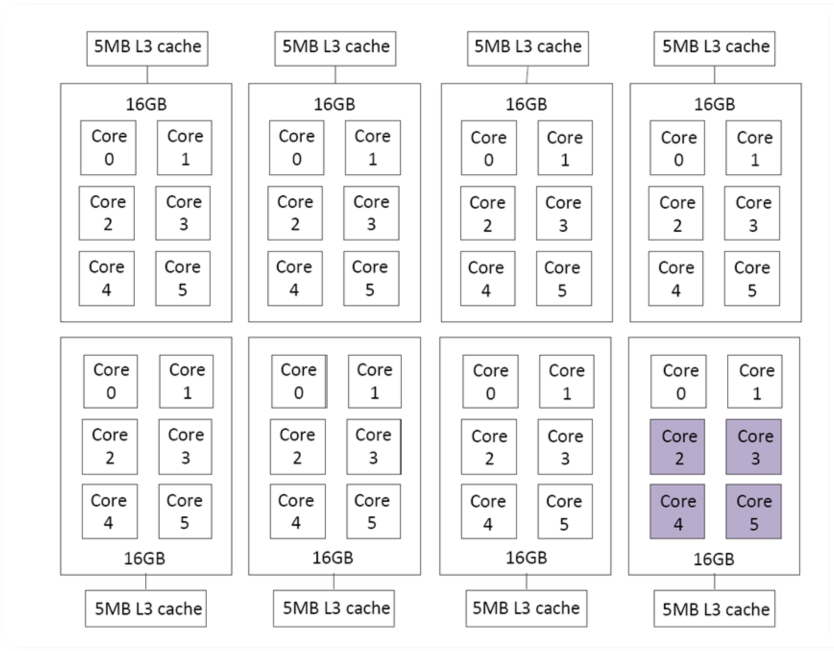
Systems@ETH zürich

- Problem:
  - Place 4 threads of a join operator
- Parameters:
  - Selectivity of join
    - Low selectivity ⇒ share caches for locality
    - High selectivity ⇒ use more caches for speed
  - Inter-cache latency
  - Size of L1 cache
  - Size of (shared) L2 cache

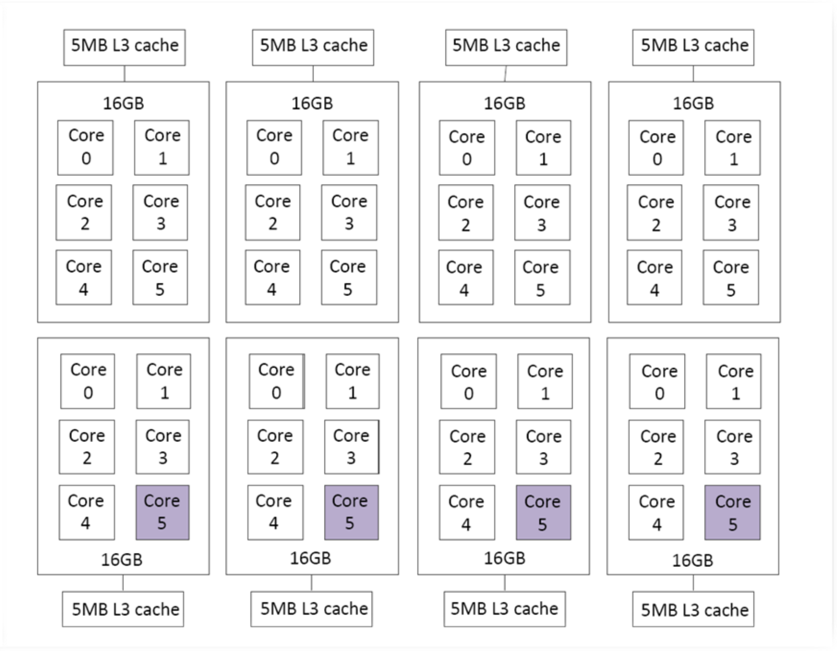
Jana Giceva,
Adrian Schüpbach,
Gustavo Alonso

Deployment suggestions on different machines

AMD Magny Cours



(a) No preference for cache-sharing

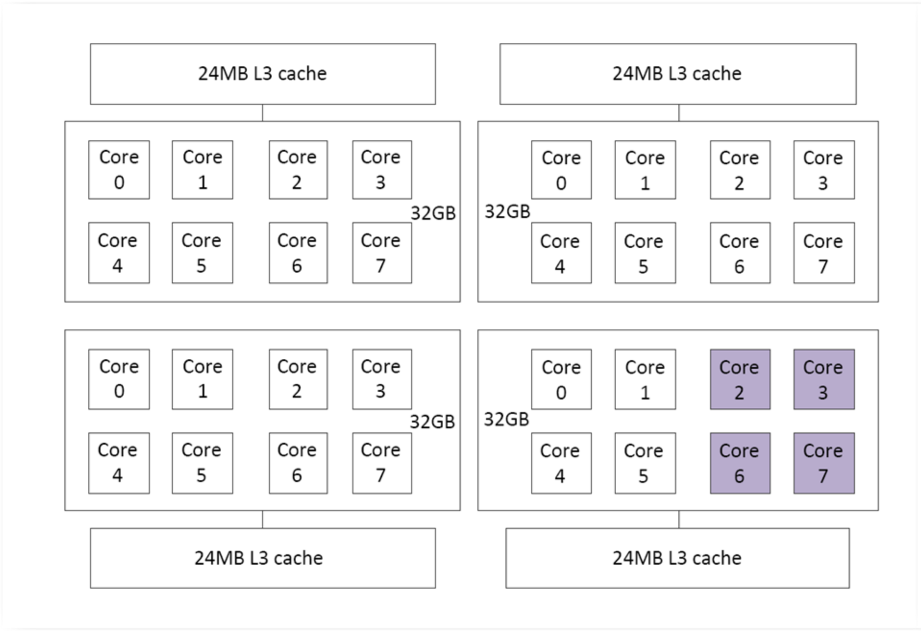


(b) No cache-sharing

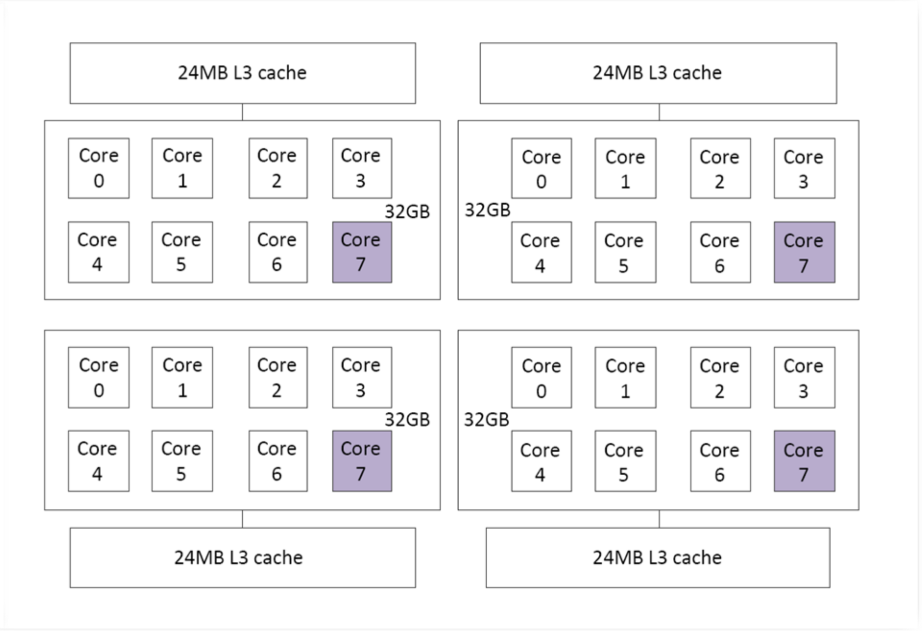
For joins: depends on selectivity of the database.

Deployment suggestions on different machines

Intel Nehalem - EX



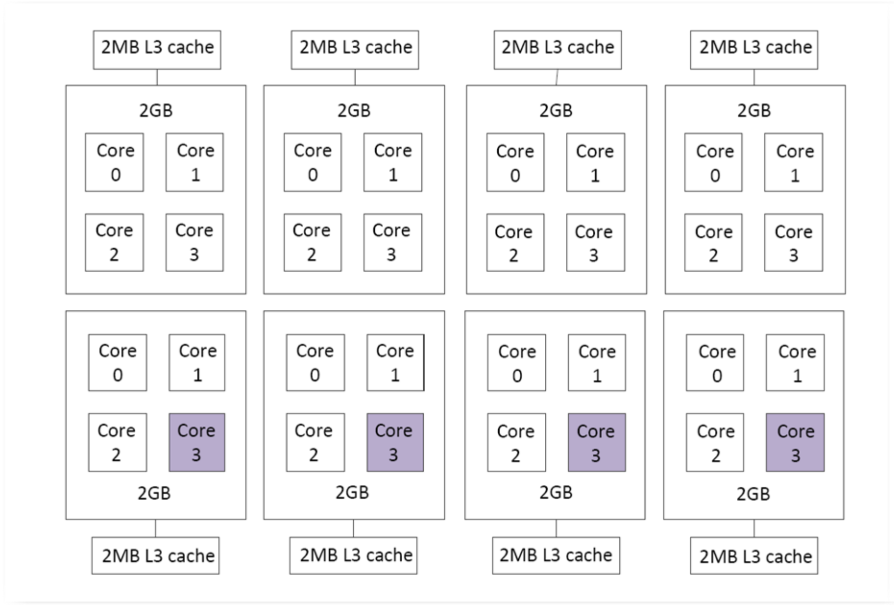
(a) No preference for cache-sharing



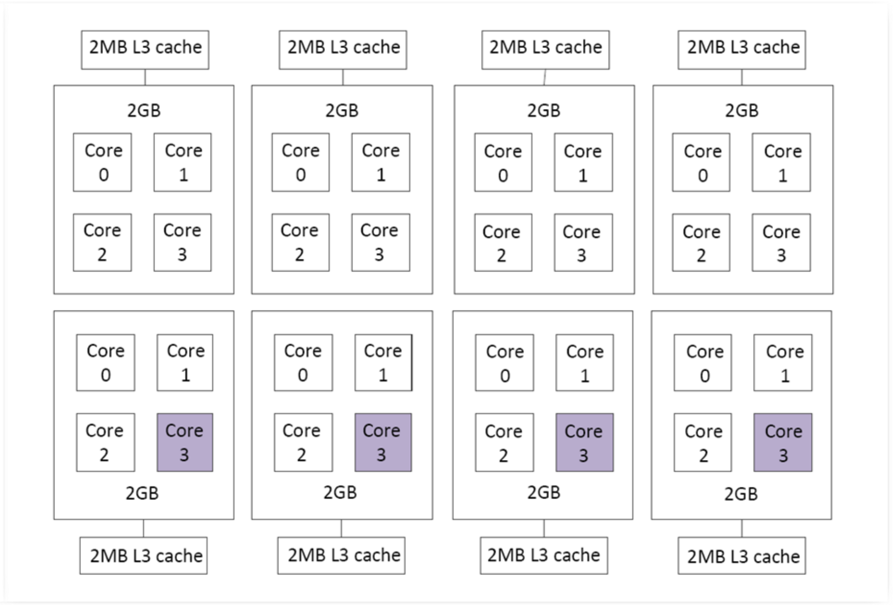
(b) No cache-sharing

Deployment suggestions on different machines

AMD Barcelona



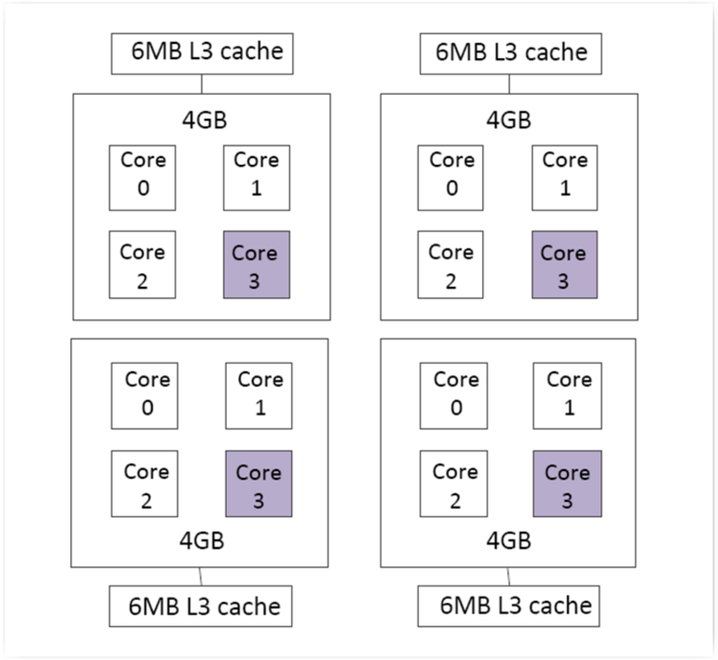
(a) No preference for cache-sharing



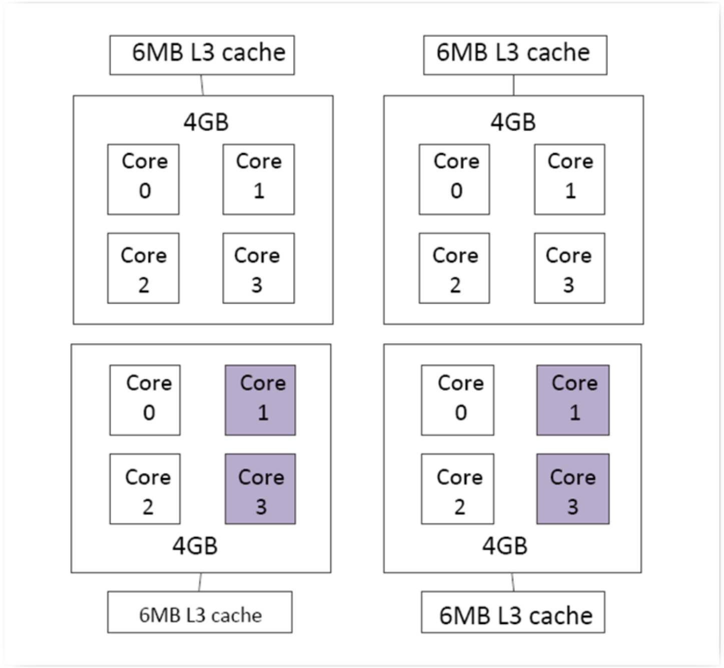
(b) No cache-sharing

Deployment suggestions on different machines

AMD Shanghai



(a) No preference for cache-sharing



(b) No cache-sharing

OK…

- We can do a better job of reasoning about hardware inside the OS.
- Simplifies programming
- Gives richer API
- Improves application performance

- But, it’s still a programming tool.

2ND TRY: FORMALLY SPECIFY SEMANTICS OF HARDWARE

Current work!

- Describe formally the hardware
  as seen by software
- Generate code, data, and proofs
  - Header files
  - SKB facts
  - Consistent (re)configuration code via
    Program Synthesis
- Long-term goal: metrics for tastefulness

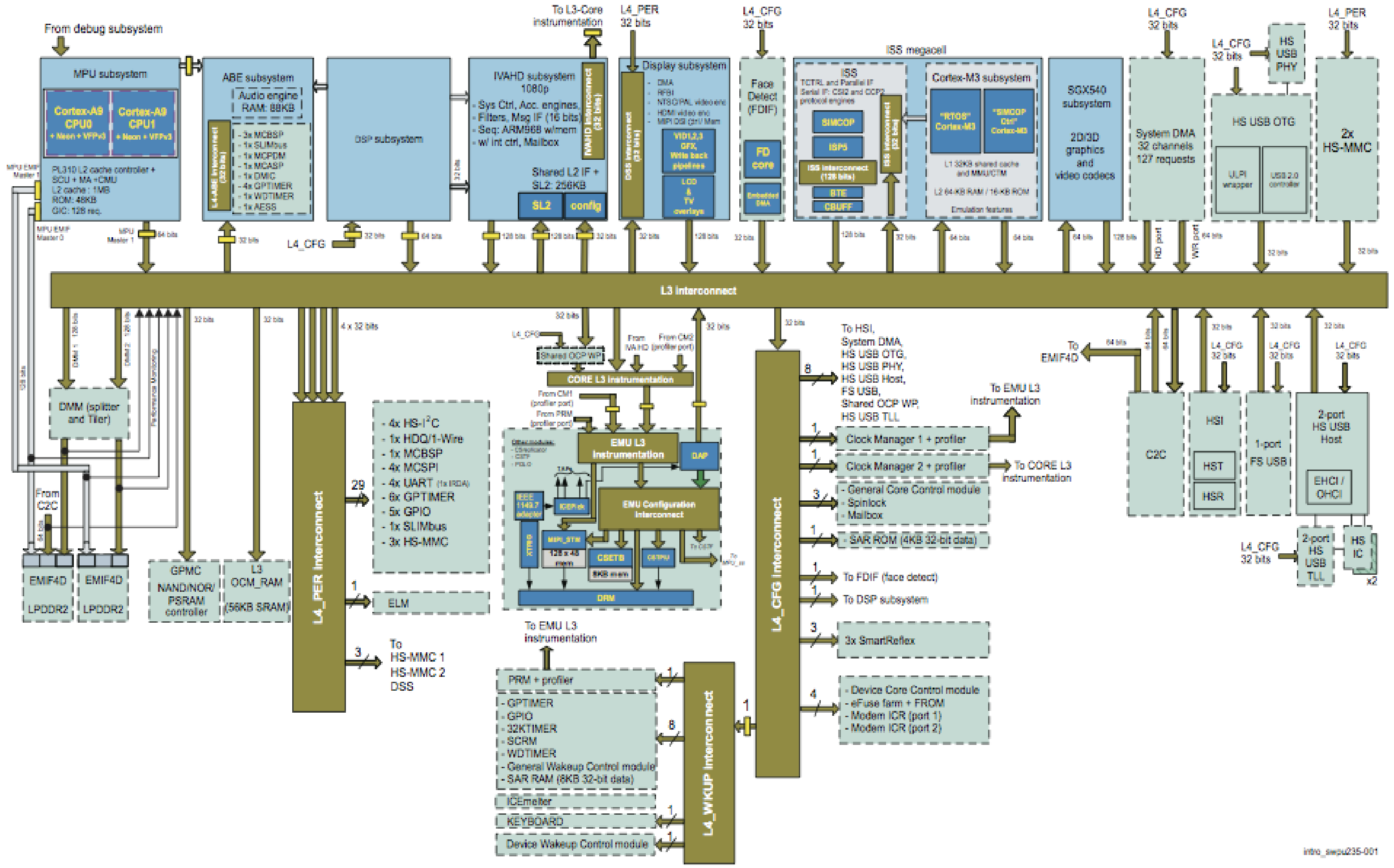
Reto Achermann,
Lukas Humbell,
David Cock

Key principles

1. Don’t **idealize** the hardware in any way.
2. Don’t **exclude** any “difficult” hardware.

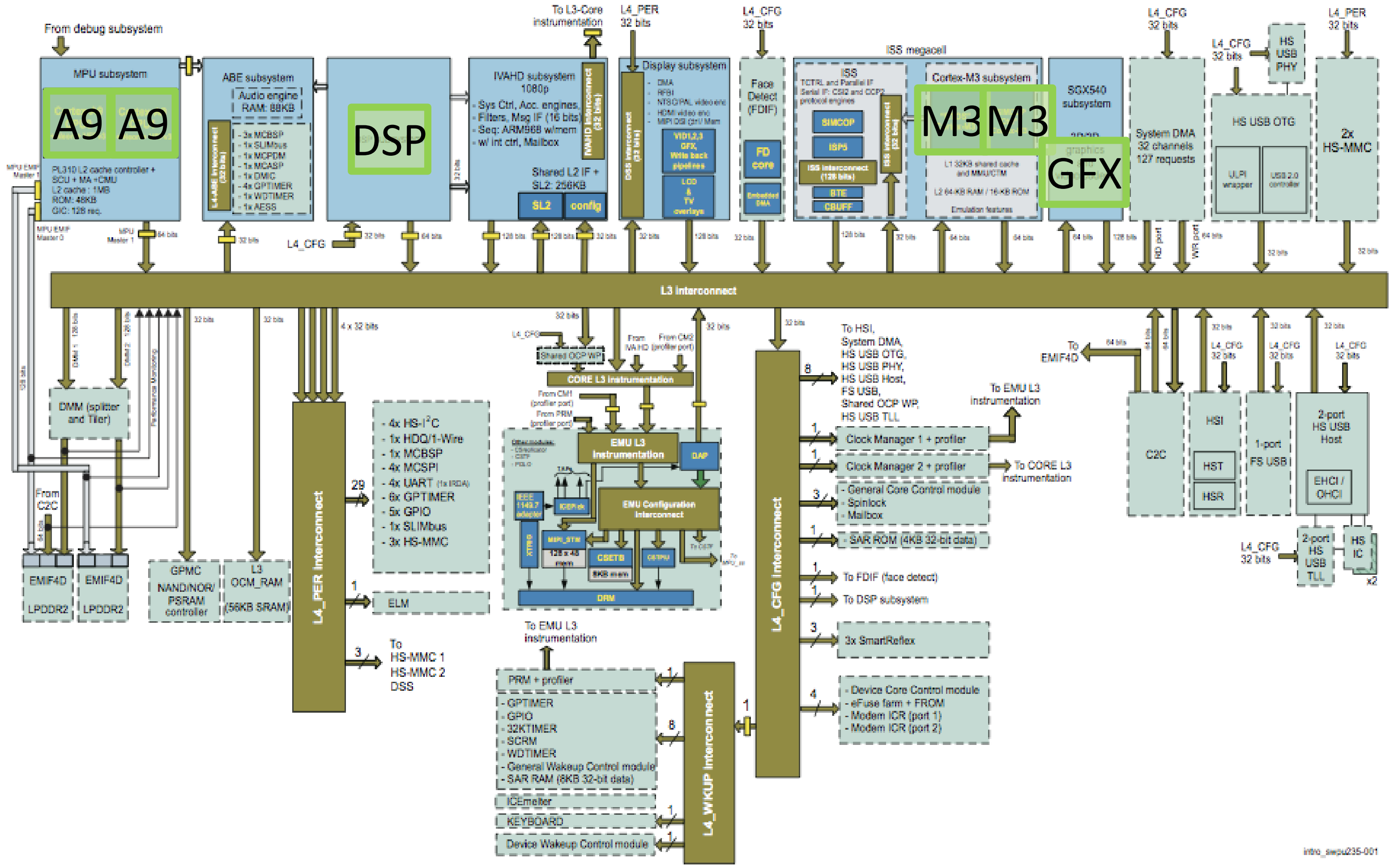
Embrace the mess, and stare into the abyss!

A closer look at the OMAP4460

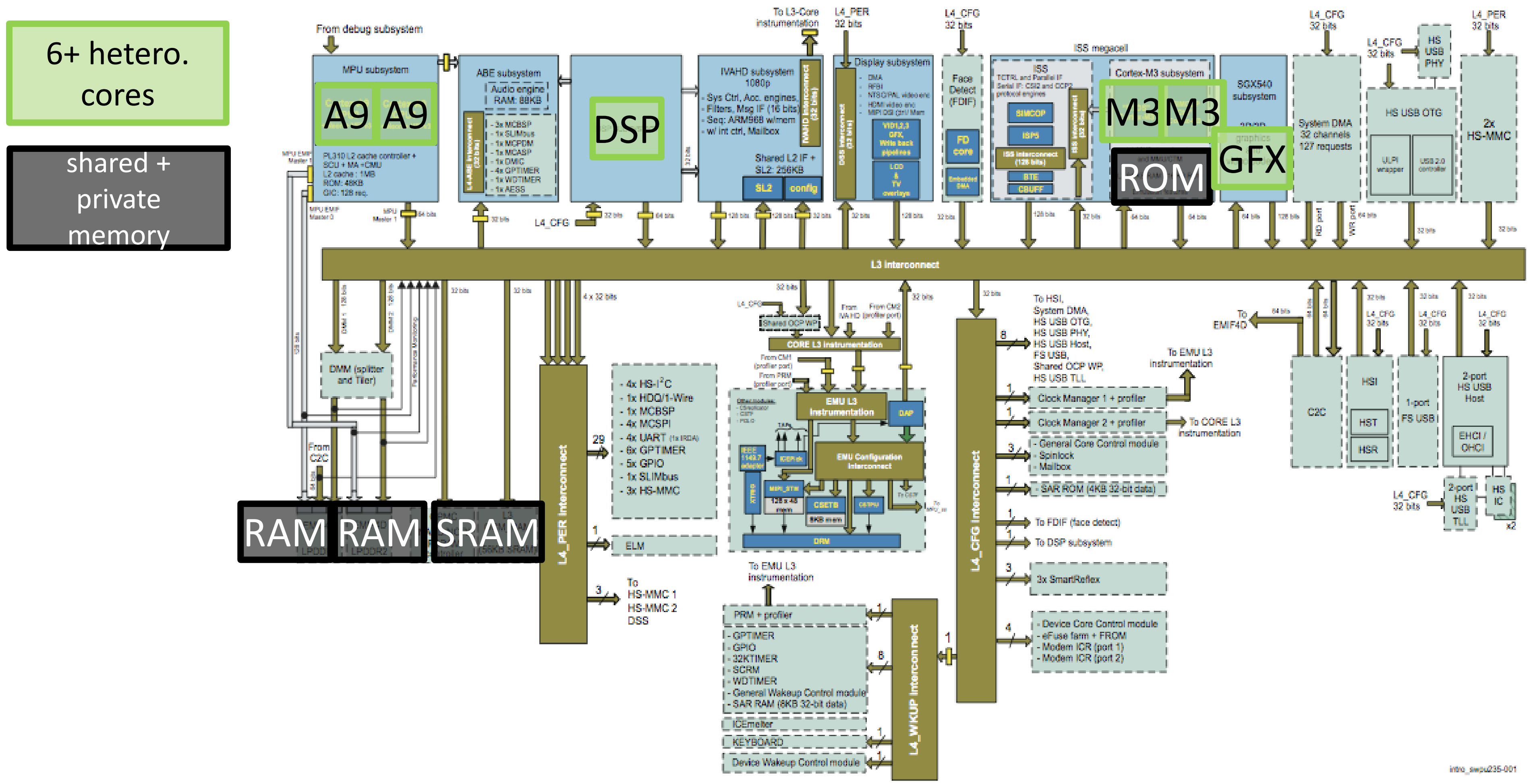


A closer look at the OMAP4460

6+ hetero. cores



A closer look at the OMAP4460



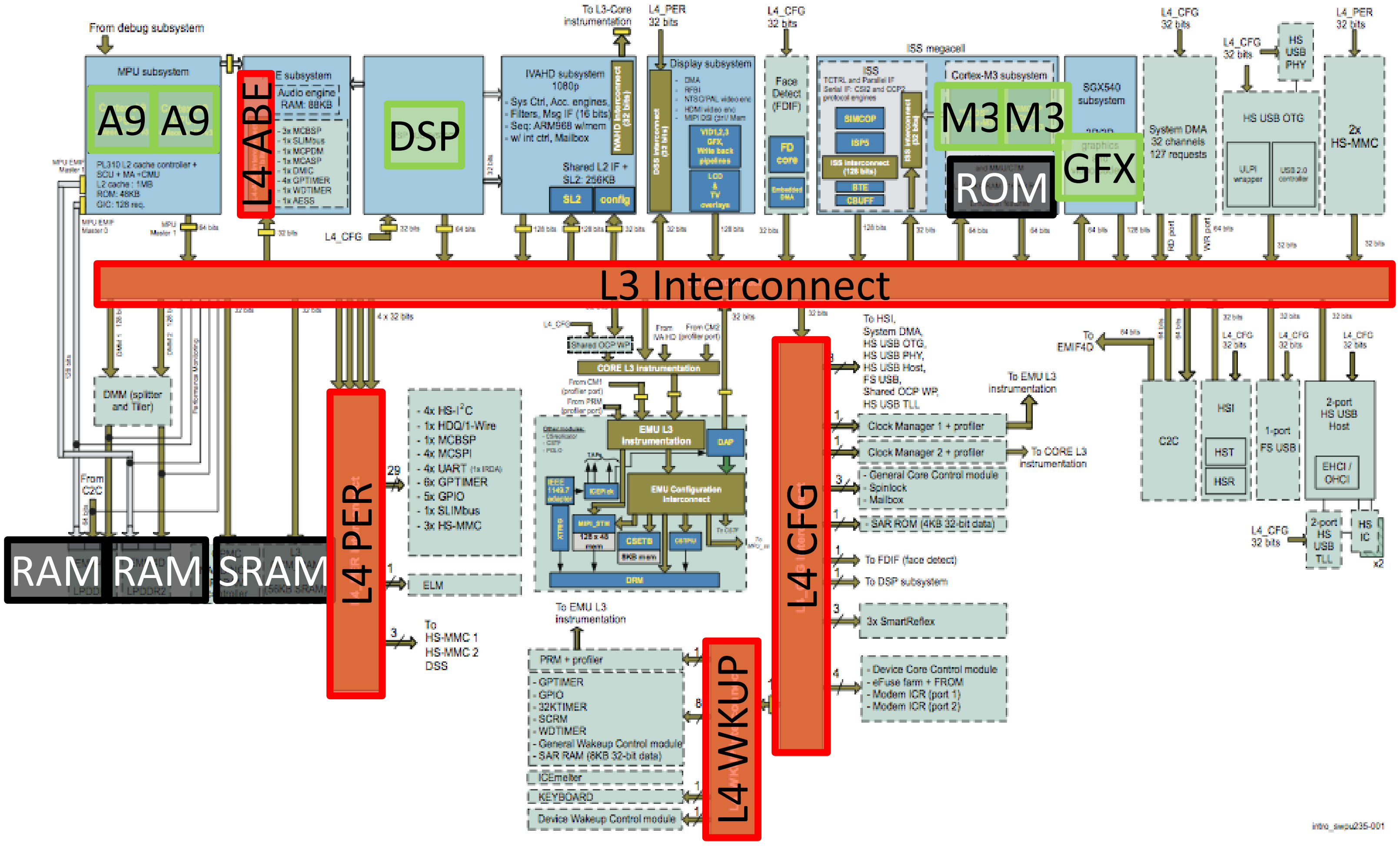
A closer look at the OMAP4460

Systems@ETH zürich

🟩 6+ hetero. cores

⬛ shared + private memory

🟥 5+ Inter-connects



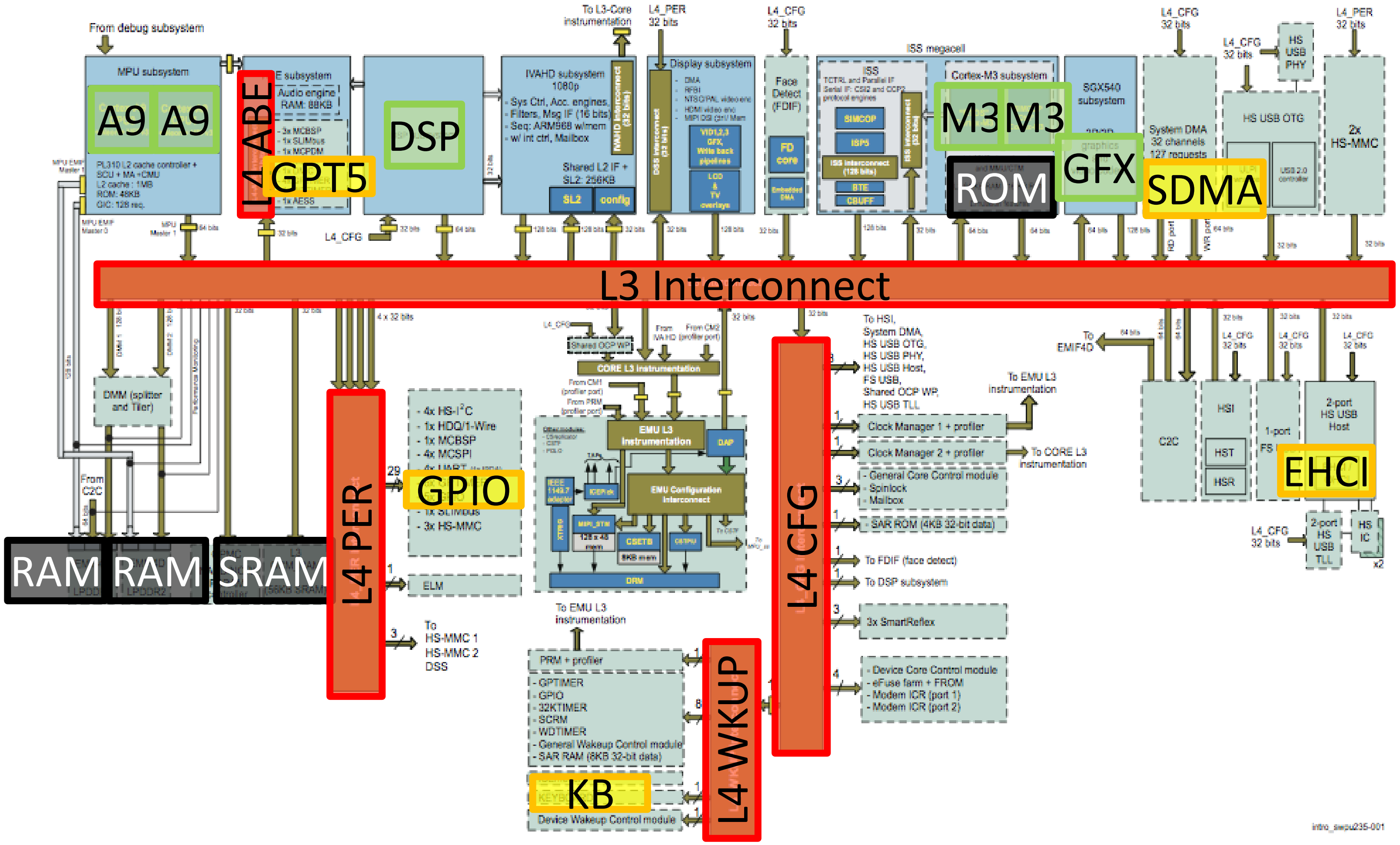
A closer look at the OMAP4460

6+ hetero. cores

shared + private memory

5+ Inter-connects

Devices on different buses



A closer look at the OMAP4460

Systems@ETH zürich

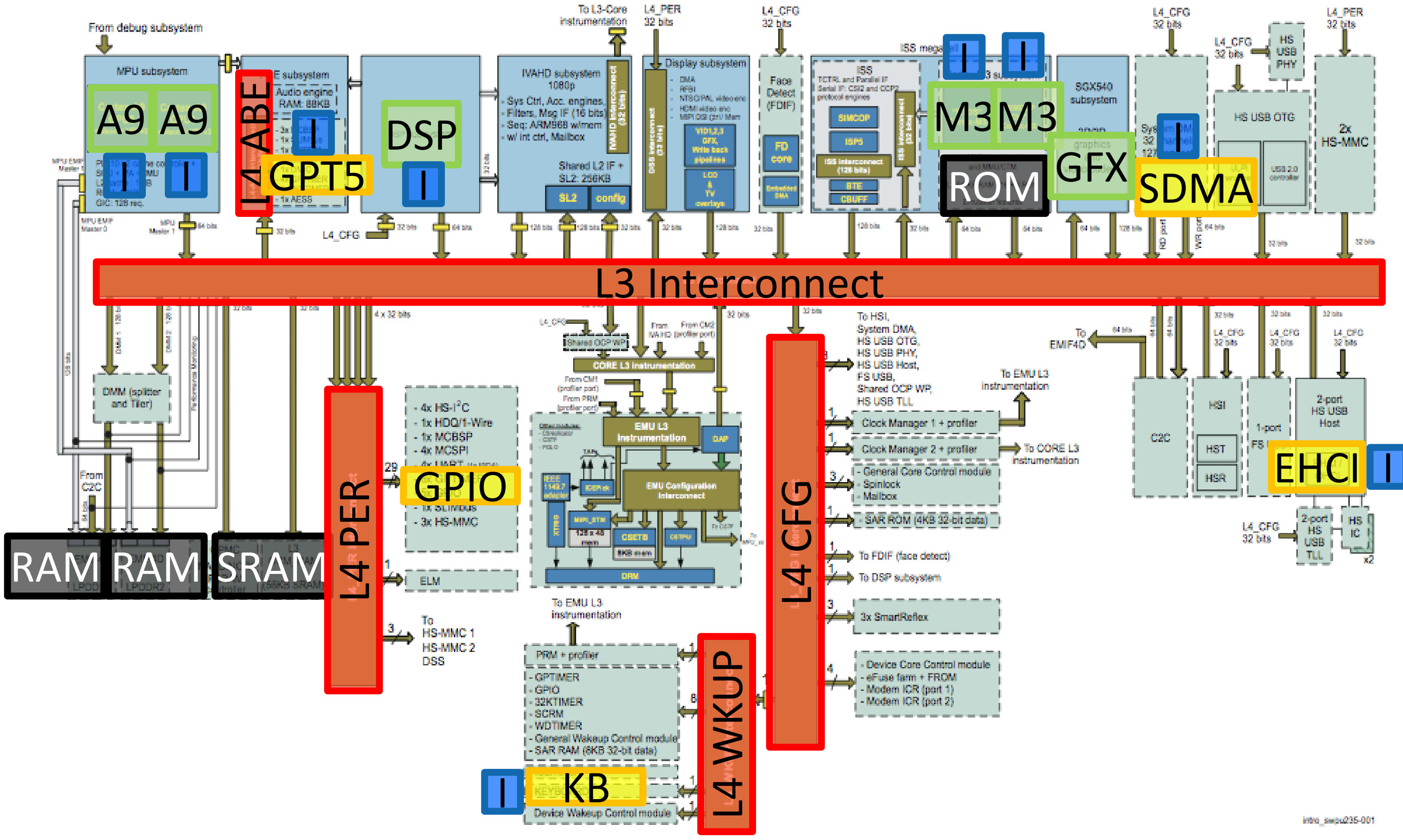
6+ hetero. cores

shared + private memory

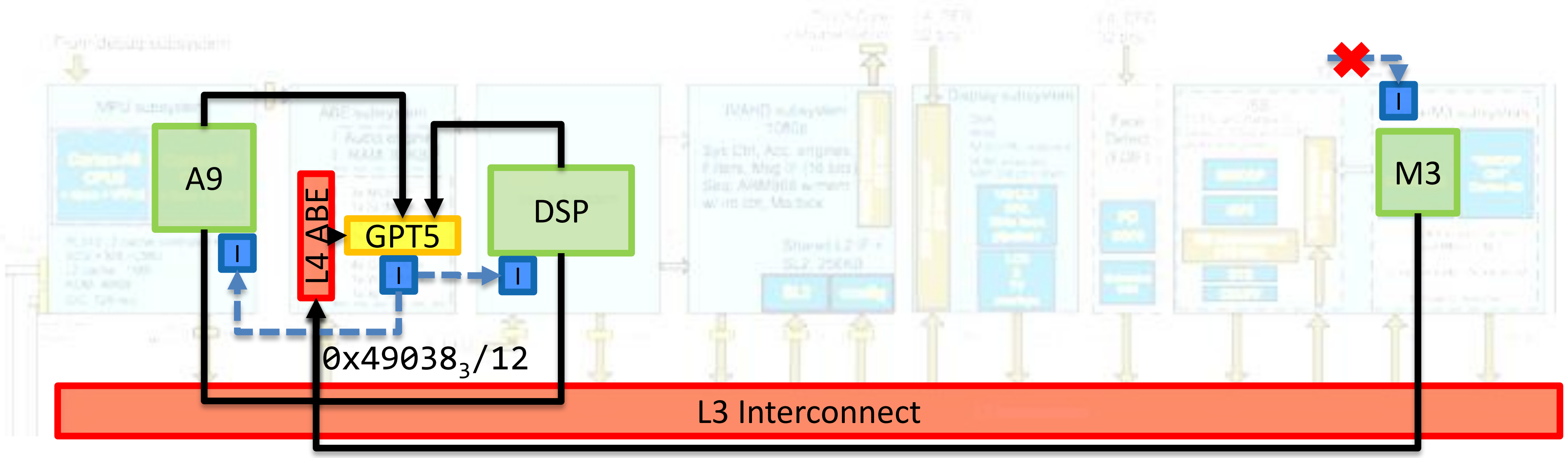
5+ Inter-connects

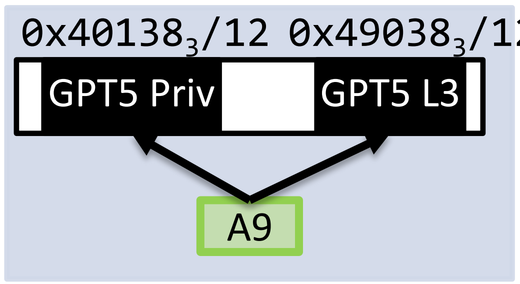
Devices on different buses

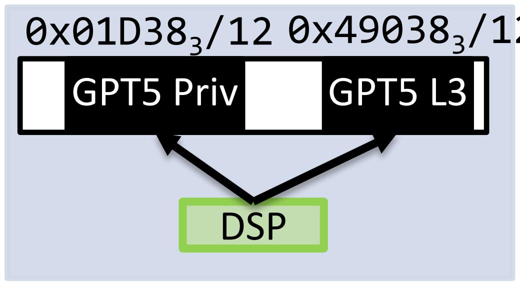
interrupt subsystem

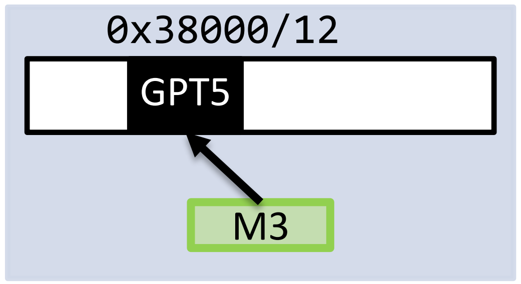


There is no uniform view of the system from all cores









Writing correct software…

… means getting all this **right**

- C code is frequently wrong.
- Nice to generate this code, but from what?

- **Proving** software correct requires a **specification** of the hardware
  - But what would it look like?
  - What could be generated from it?

Model addresses as a *decoding net*

net$_s$:= [ℕ **is** node$_s$, ℕ.. ℕ **are** node$_s$, ...]

node$_s$:= (**accept** [block$_s$, ...])? (**map** [map$_s$, ...])? (**over** N)?

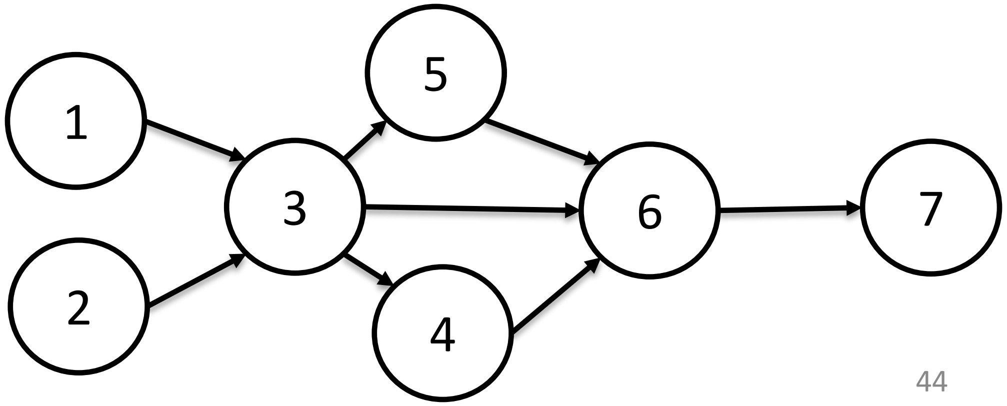
map$_s$:= block$_s$ **to** [ℕ (**at** ℕ)?, ...]

block$_s$:= ℕ – ℕ

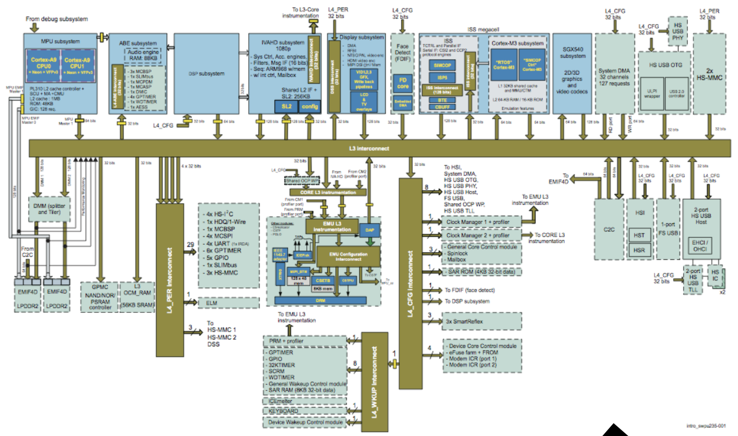
Overlays another node (1-to-1 mapping)

Translates a block of addresses to a set of nodes at potential different addresses

Note: can be cyclic!



Representing address decoding



$V_{A9:0}$ **is map** $[\mathtt{20000}_3/12$ **to** $P_{A9:0}$ **at** $\mathtt{80000}_3]$

$P_{A9:0}, P_{A9:1}$ **are map** $[\mathtt{40138}_3/12$ **to** GPT **at** $\mathtt{0}]$ **over** $L3$

P_{DSP} **is map** $[\mathtt{1d3e}_3/12$ **to** GPT **at** $\mathtt{0}]$ **over** $L3$

V_{M3}, V_{M3} **are over** $L1_{M3}$

RAM_{M3} **is accept** $[\mathtt{55020}_3/16]$

ROM_{M3} **is accept** $[\mathtt{55000}_3/14]$

$V_{A9:1}$ **is map** $[\mathtt{20000}_3/12$ **to** $P_{A9:1}$ **at** $\mathtt{80000}_3]$

V_{DSP} **is over** P_{DSP}

$L2_{M3}$ **is map** $[\mathtt{0}_{30}$ **to** $L3$ **at** $\mathtt{80000}_3]$

$L1_{M3}$ **is map** $[\mathtt{0}_{28}$ **to** $MIF]$

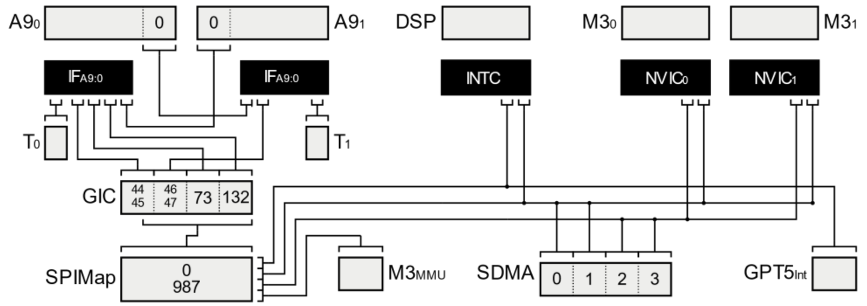
$L4$ **is map** $[\mathtt{49038}_3/12$ **to** GPT **at** $\mathtt{0}]$

GPT **is accept** $[\mathtt{0}/12]$

MIF **is map** $[\mathtt{0}-\mathtt{5ffffff}$ **to** $L2_{M3}, \mathtt{55000}_3/14$ **to** $RAM_{M3}, \mathtt{55020}_3/16$ **to** $ROM_{M3}]$

$L3$ **is map** $[\mathtt{49000}_3/24$ **to** $L4$ **at** $\mathtt{40100}_3, \mathtt{55000}_3/12$ **to** $MIF]$ **accept** $[\mathtt{80000}_3/30]$

Interrupts actually the same

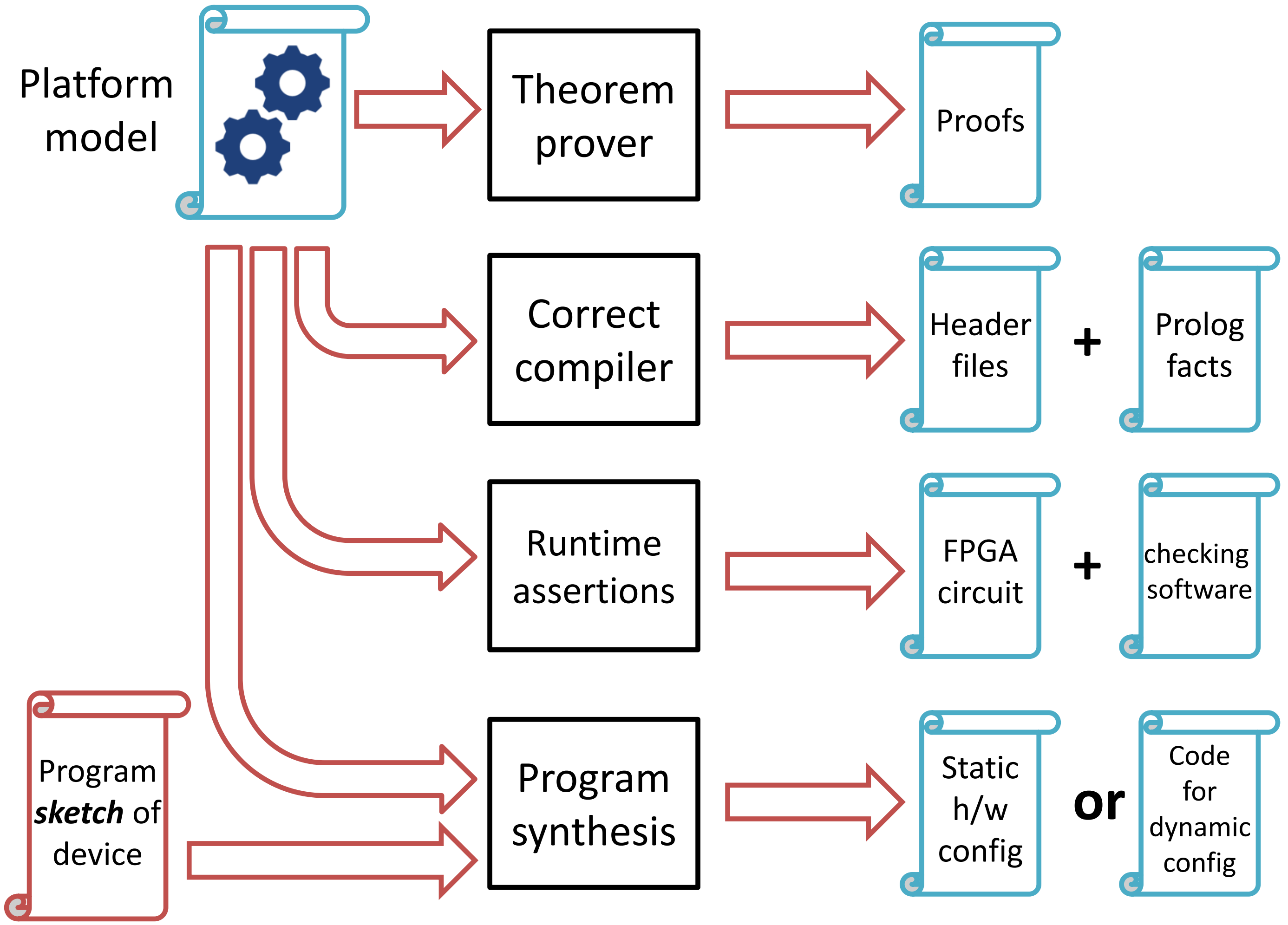


$SDMA$ **is map** [0 **to** $SPIMap$ **at** 12 **to** $INTC$ **at** 18 **to** $NVIC_0$ **at** 18 **to** $NVIC_1$ **at** 18,
1 **to** $SPIMap$ **at** 13 **to** $INTC$ **at** 19 **to** $NVIC_{M3:0}$ **at** 19 **to** $NVIC_{M3:1}$ **at** 19,
2 **to** $SPIMap$ **at** 14 **to** $NVIC_0$ **at** 20 **to** $NVIC_1$ **at** 20,
3 **to** $SPIMap$ **at** 15 **to** $NVIC_0$ **at** 21 **to** $NVIC_1$ **at** 21]
$GPT5_{Int}$ **is map** [0 **to** $SPIMap$ **at** 41 **to** $INTC$ **at** 41]
GIC **is map** [44 – 45 **to** $IF_{A9:0}$ **at** 44, 46 – 47 **to** $IF_{A9:1}$ **at** 46, . . .]
$A9_0$ **is map** [0 **to** $IF_{A9:1}$ **at** 0] $A9_1$ **is map** [0 **to** $IF_{A9:0}$ **at** 0]
T_0 **is map** [0 **to** $IF_{A9:0}$ **at** 29] T_1 **is map** [0 **to** $IF_{A9:1}$ **at** 29]
$M3_{MMU}$ **is map** [0 **to** $SPIMap$ **at** 100] $SPIMap$ **is map** [0 – 987 **to** GIC **at** 32]
$INTC, NVIC_*$ **are accept** [] IF_* **are accept** [0 – 1020]

Can capture functionality of:

- Cores and DMA engines
- Caches (both physical and virtual)
- Firewalls
- MMUs and IOMMUs
- Lookup tables
- Interrupt controllers
- Virtualization hardware

What can you do?



Long-term goal

- A language for describing hardware platforms
  - c.f. ARM's specification language
- Assemble descriptions of many devices
- Devise complexity metrics
- Create a style guide for hardware designers

What is "tasteful hardware design"?

THE BIGGER TROUBLE WITH HARDWARE

A deadly embrace

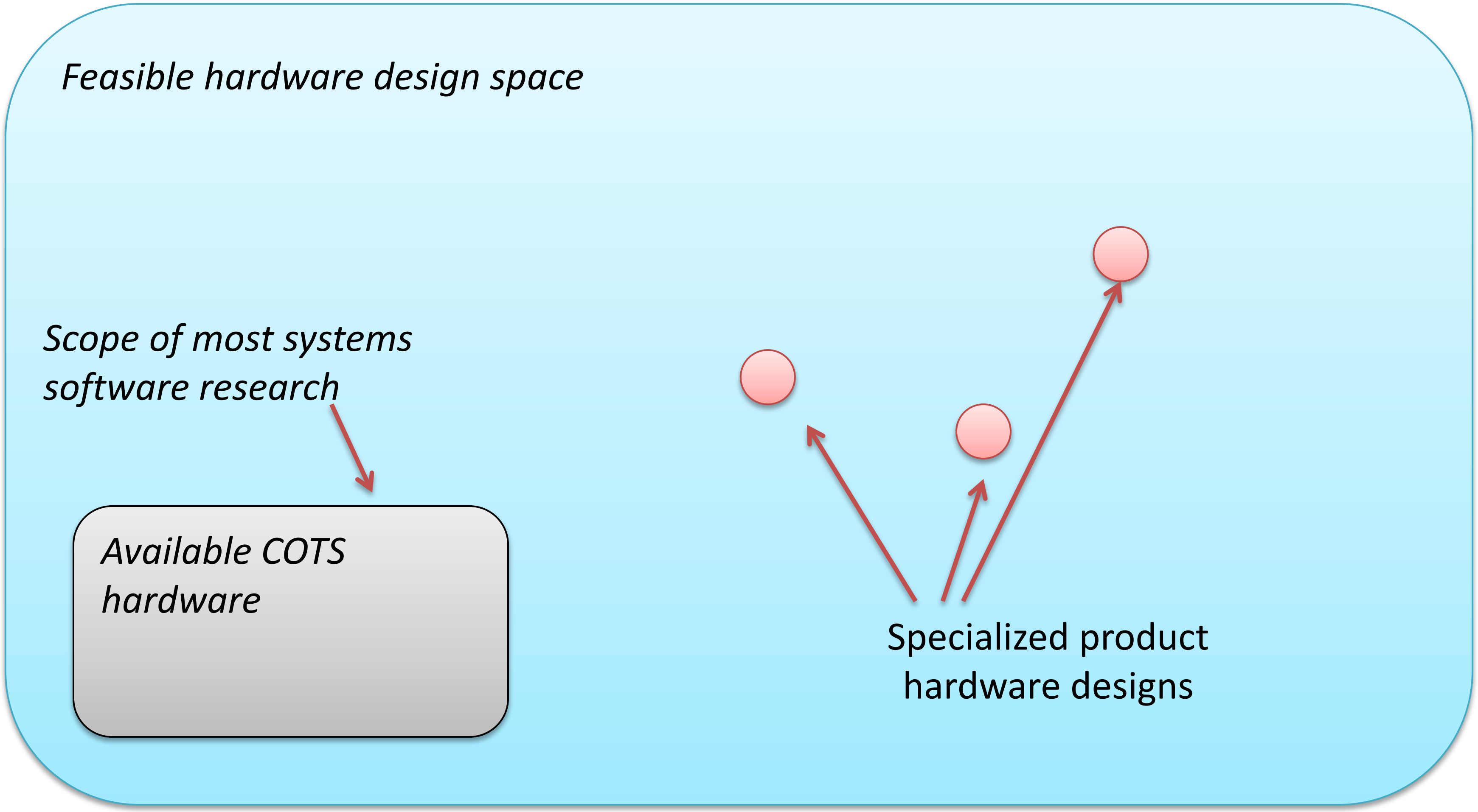
Commodity hardware is designed for current, conventional application workloads over Linux.

Academic research (and industrial innovation) in system software is constrained by available commodity hardware.

But hardware is easy to build

- Hardware is so complex and diverse
  because it's so easy to build what you want
  - High-end CAD systems
  - Simulators and emulators, FPGAs
  - Rapid fabrication of boards and ASICs
- Big companies ***do***
  - HPE's The Machine
  - Oracle RAPID, SPARC M7
  - Amazon F1
  - Microsoft Catapult
  - Google TPU for Tensorflow

Challenge for research



Feasible hardware design space
Scope of most systems software research
Available COTS hardware
Specialized product hardware designs

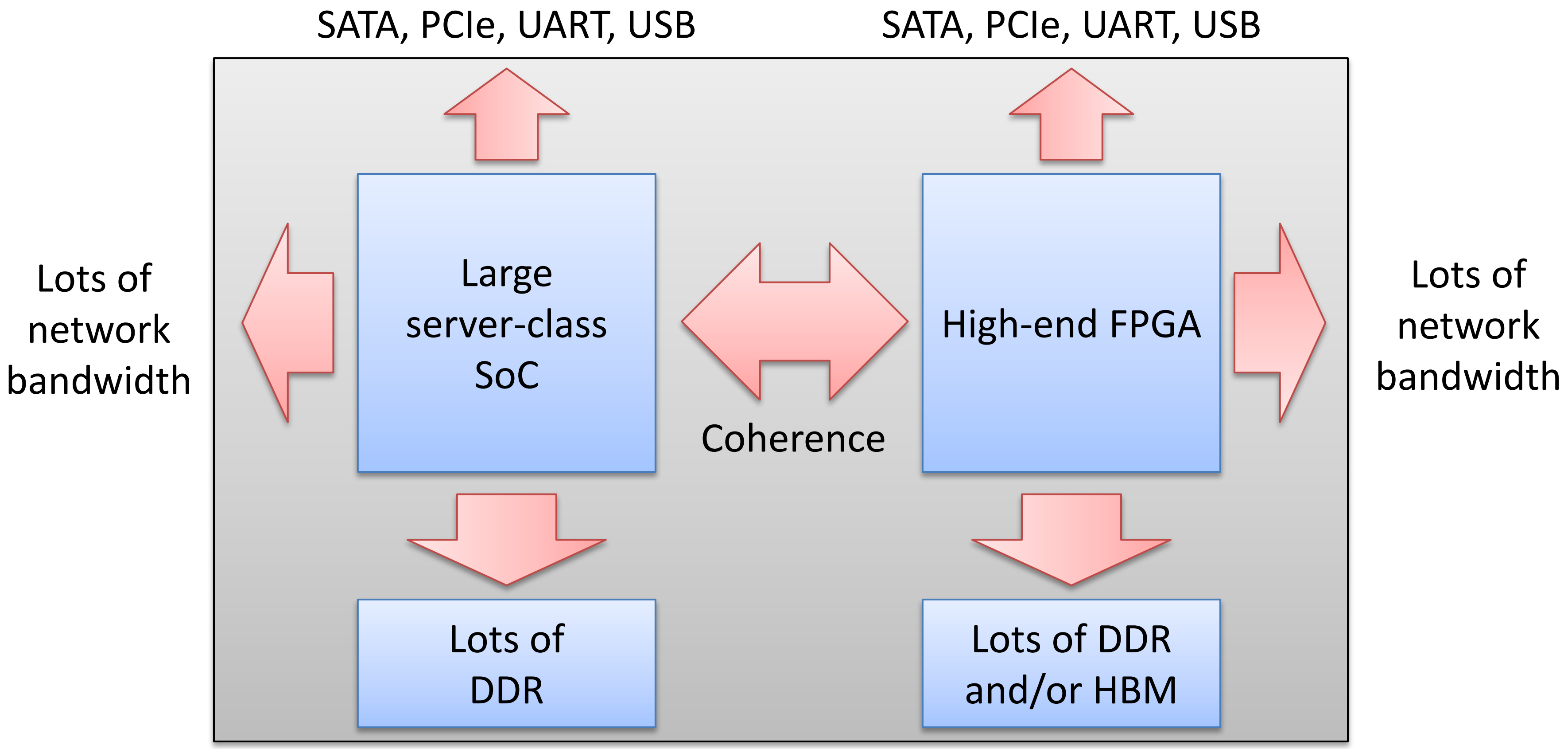
SO, WHAT CAN WE DO?

ENZIAN

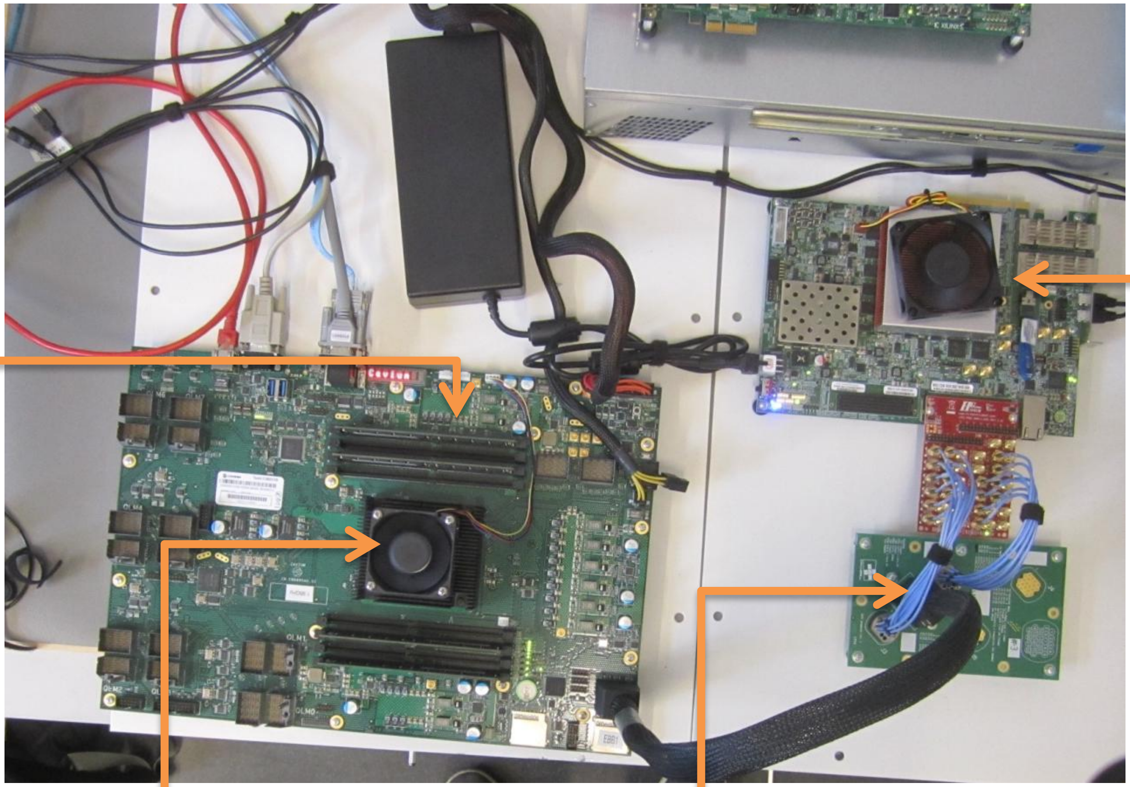
What if we had…

- A hardware *research* platform for system software
  - Massively *overengineered* wrt. products
  - Highly *configurable* building block for rackscale
- Perhaps we can *actually build it* at ETH…
  - Logical next platform for our research
  - Seed to other universities for impact

Sketch: the basic building block

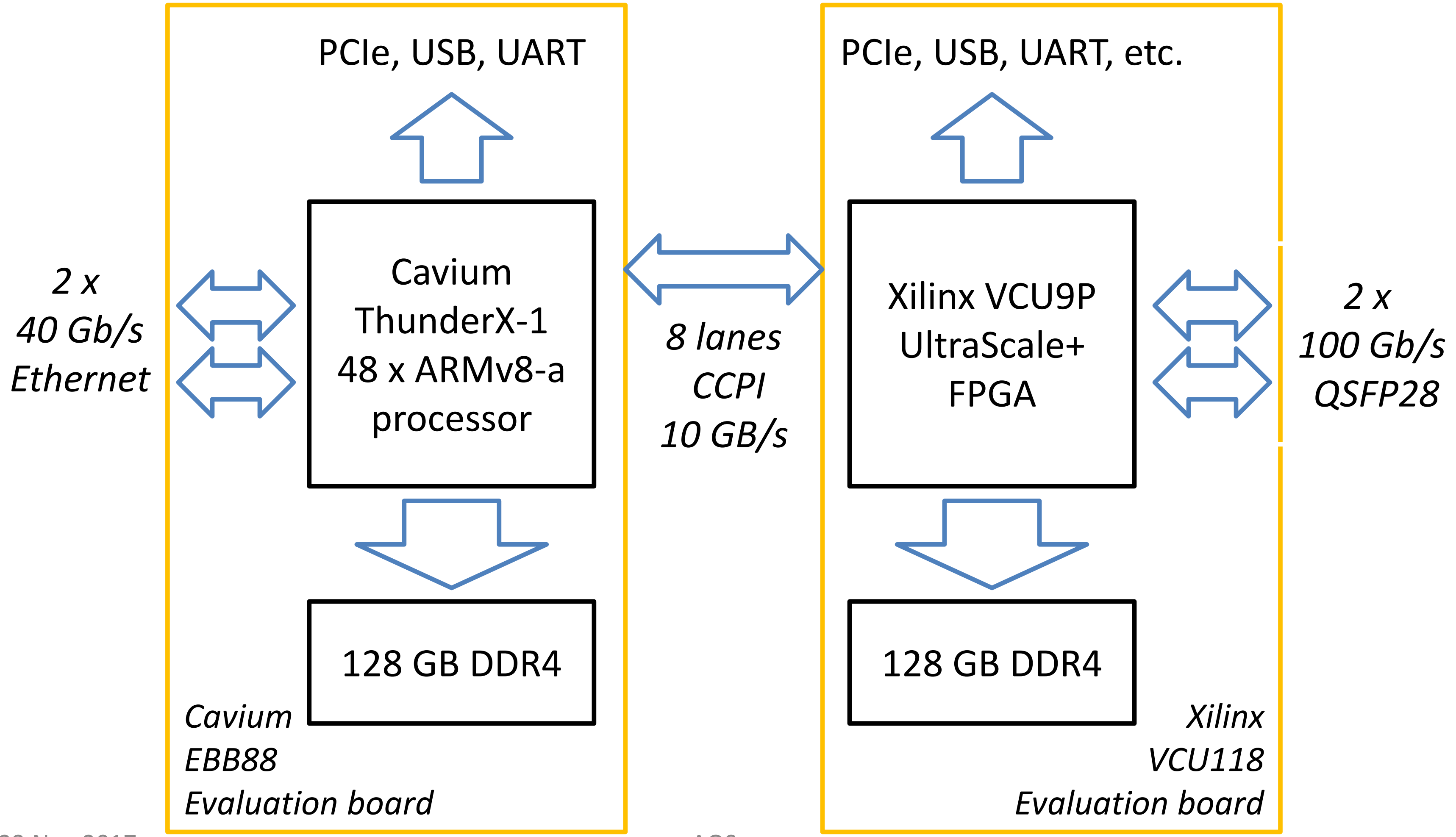


Enzian v.1



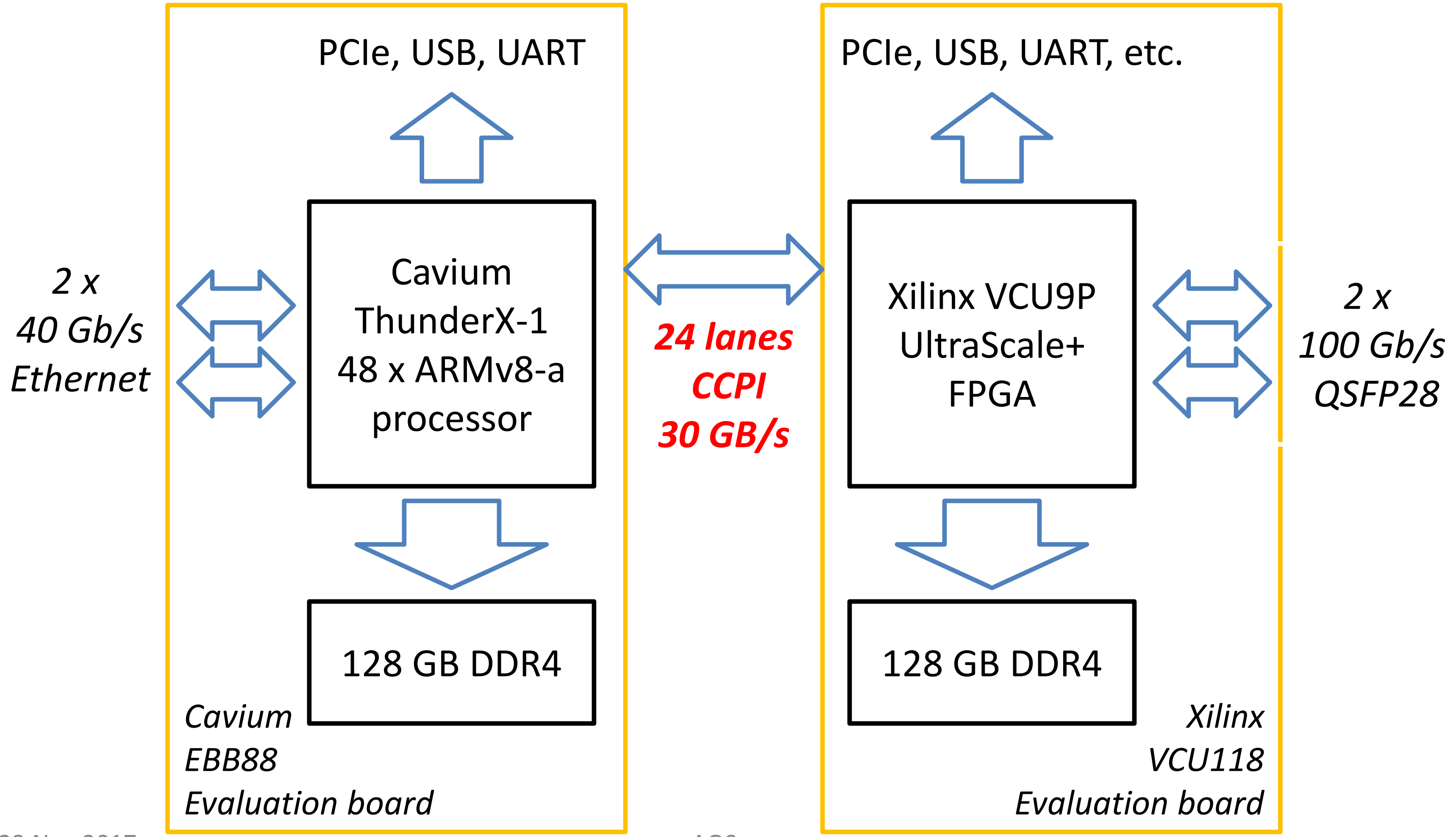
1 x Coherence link (80Gb/s)

Enzian v.1

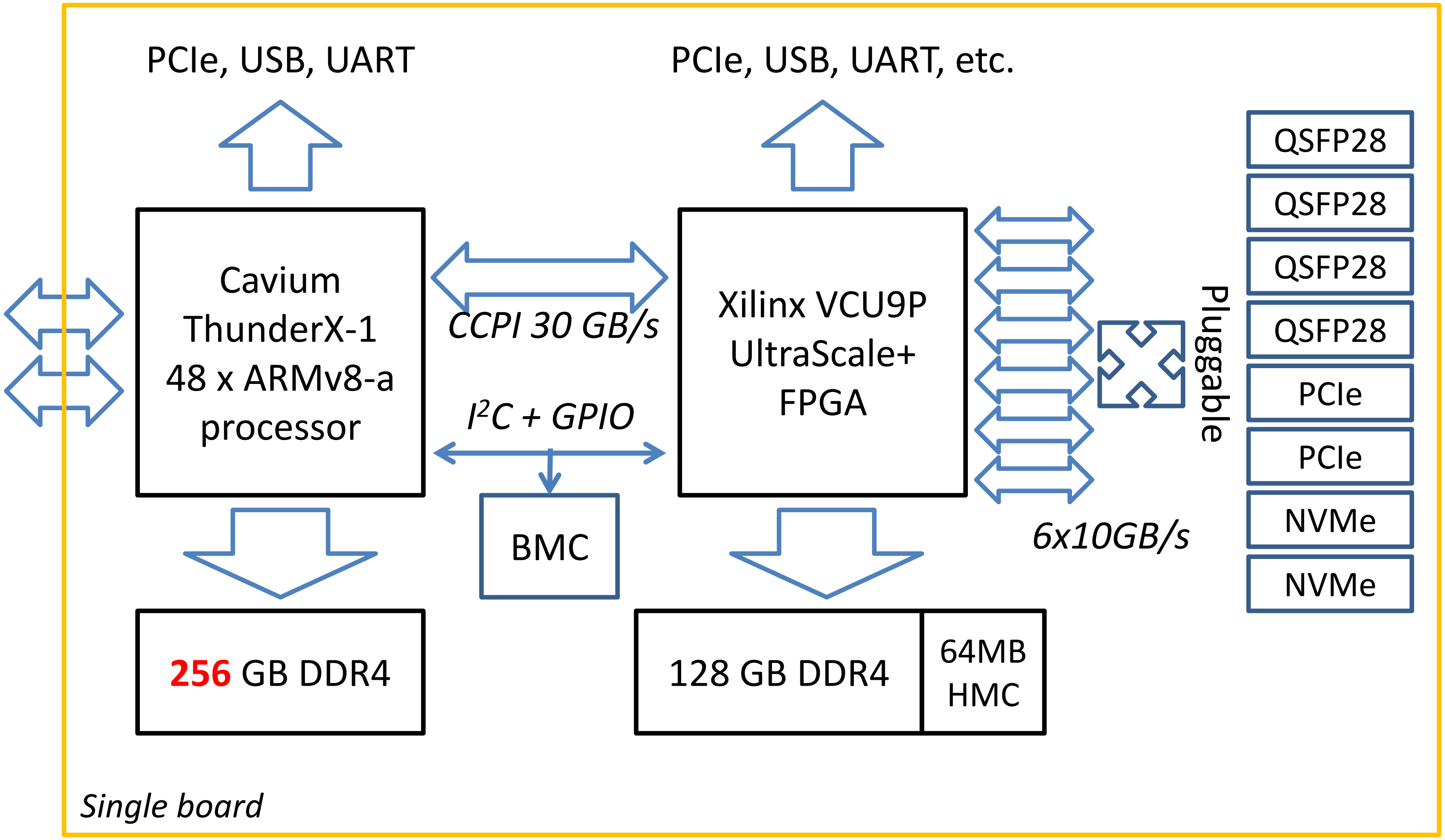


2 x
40 Gb/s
Ethernet
PCIe, USB, UART
Cavium
ThunderX-1
48 x ARMv8-a
processor
128 GB DDR4
Cavium
EBB88
Evaluation board
8 lanes
CCPI
10 GB/s
PCIe, USB, UART, etc.
Xilinx VCU9P
UltraScale+
FPGA
128 GB DDR4
Xilinx
VCU118
Evaluation board
2 x
100 Gb/s
QSFP28

Enzian v.2 (November 2017)


PCIe, USB, UART
PCIe, USB, UART, etc.
2 x 40 Gb/s Ethernet
Cavium ThunderX-1 48 x ARMv8-a processor
24 lanes CCPI 30 GB/s
Xilinx VCU9P UltraScale+ FPGA
2 x 100 Gb/s QSFP28
128 GB DDR4
128 GB DDR4
Cavium EBB88 Evaluation board
Xilinx VCU118 Evaluation board

Enzian v.3 (2018)



All kinds of uses for this…

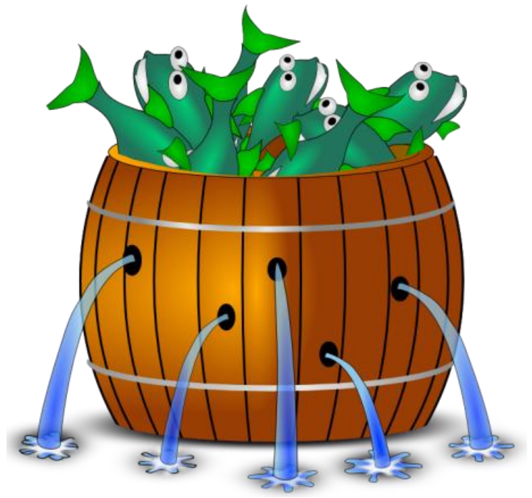
- Plug lots together for rack-scale computing
- Use the FPGA for data processing offload
- A better NetFPGA, or bump in the wire
- FPGA support infrastructure
- Sequester processors using the FPGA
- Runtime verification of program trace
- Experiment in scaling coherency

etc.

You can probably think of more…

Summary 1: Hardware is easy to build

- It is **complex**, **diverse**, and **changes** rapidly
- It is hard to program in C
- It has totally **unspecified** semantics
- OS researchers need to up our game



Summary 2:
COTS hardware is unrealistic

Systems@ETH Zürich

- All the product action uses **custom** hardware
- Vendors can build almost **anything**
- What to build is an **economic** question
  - $\Rightarrow$ not something we can answer
- We needs **overengineered research platforms**
  - Our goal should be to deliver **options** and **techniques**

ENZIAN

www.enzian.systems

Many thanks!