ETH zürich

The Impact of Incomprehensible Hardware on Security

Systems@ETH Zürich

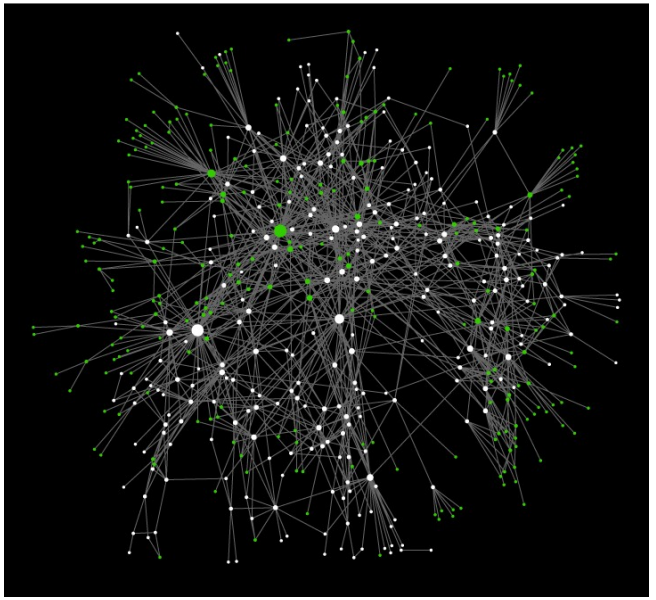
Telling a Story

- We don’t really understand hardware (and it hurts security). Examples from seL4:
  - Undocumented hardware bugs
  - Side channels
- We’re trying to fix this:
  - Formal hardware models to drive OS actions
  - Runtime verification
  - Building better hardware

seL4

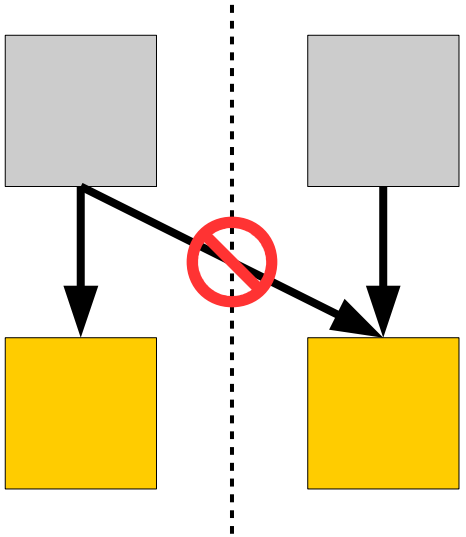
seL4 is a verified, high-performance microkernel with:

- Proven functional correctness
- Proven authority confinement
- Proven information flow

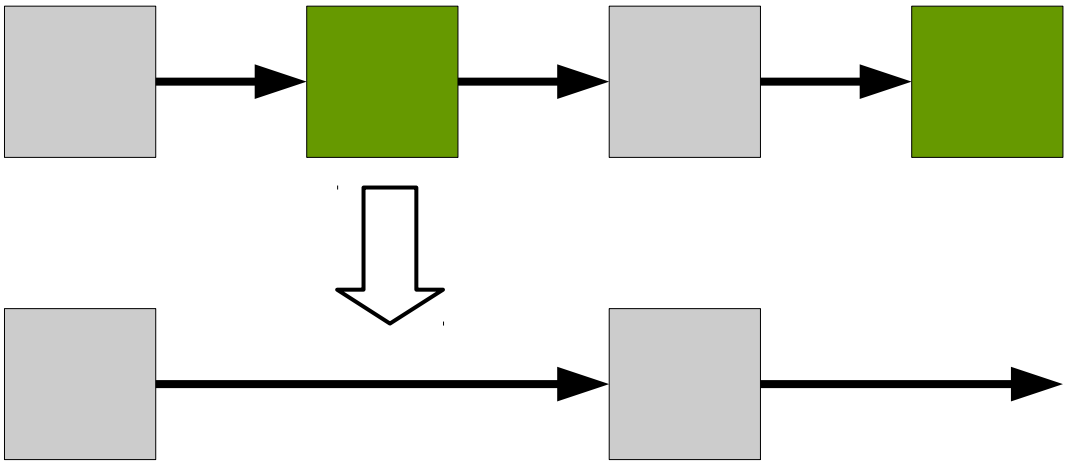


High-Level Properties

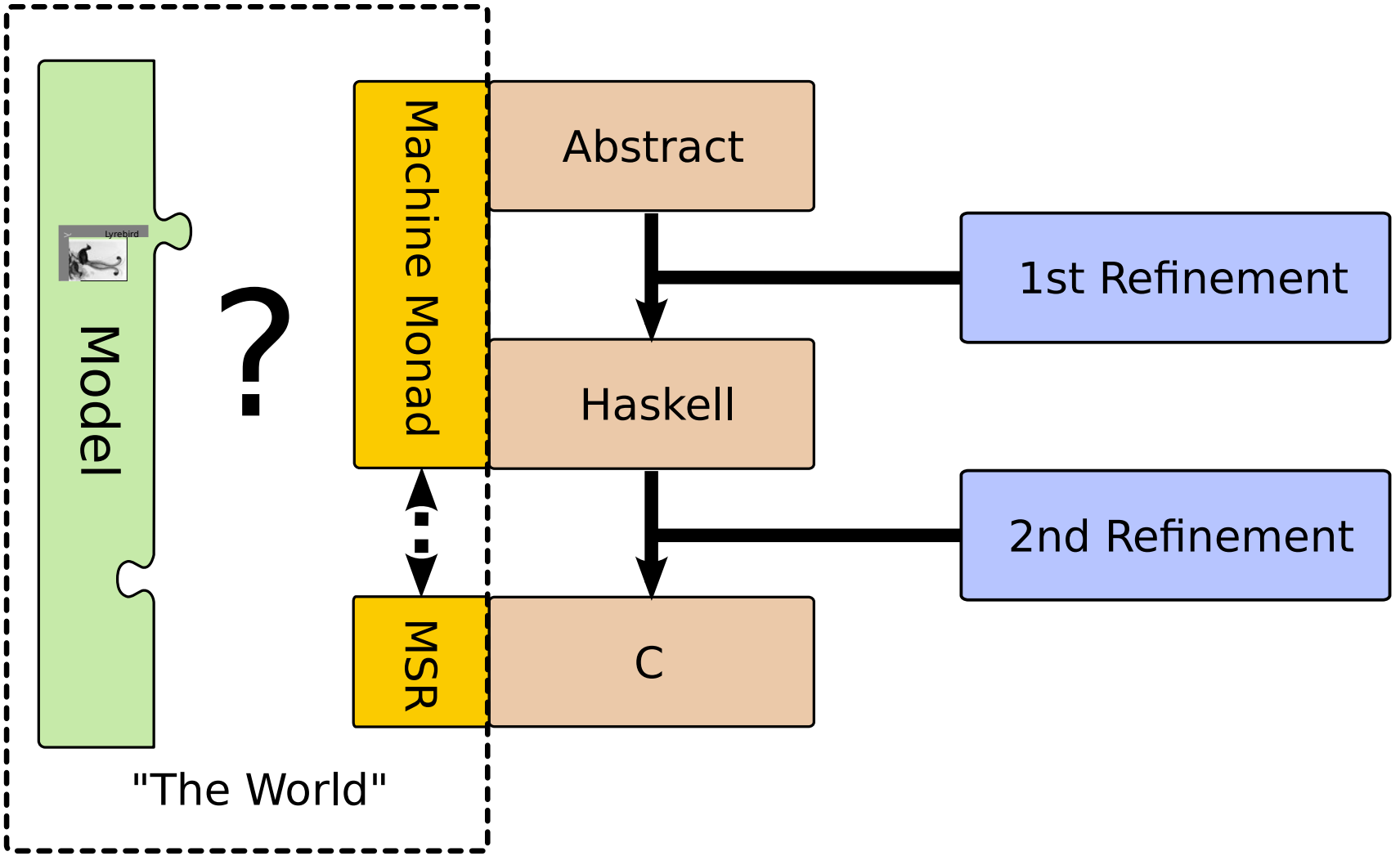
Confinement



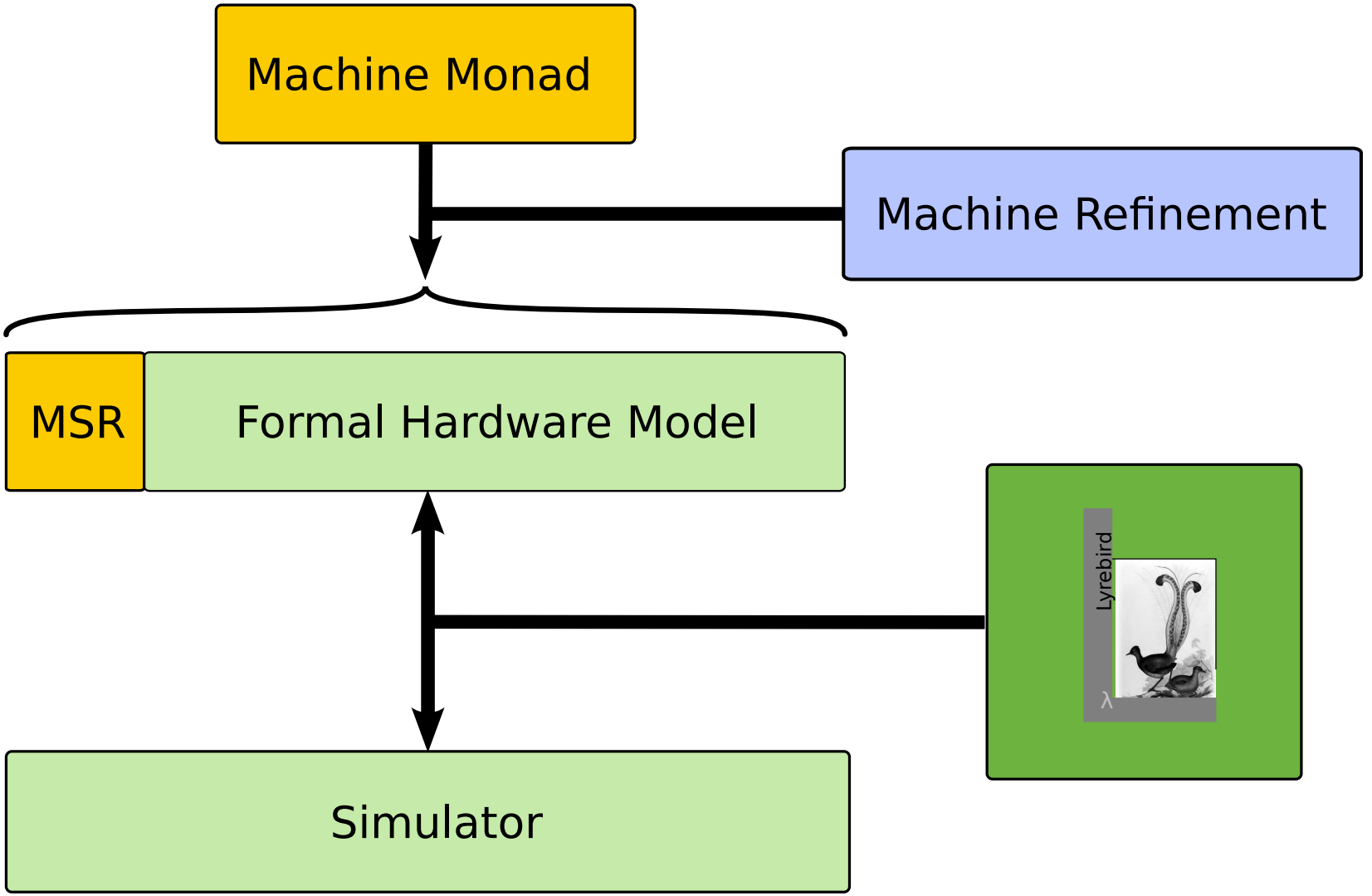
Noninterference



The Structure of the Proof



The seL4 Machine Model



A Worked Example

What does this code do? What ends up in r1?

address	data	instruction	r1	r2	r3	@100	@108
...	...	...	...	100	108	42	...
1000	e5921000	ldr r1, [r2]					
1004	e5832000	str r1, [r3]					
1008	e2811001	add r1, r1, #1					

A Worked Example

What does this code do? What ends up in r1?

address	data	instruction	r1	r2	r3	@100	@108
...	...	...	...	100	108	42	...
1000	e5921000	ldr r1, [r2]	42	100	108	42	...
1004	e5832000	str r1, [r3]					
1008	e2811001	add r1, r1, #1					

A Worked Example

What does this code do? What ends up in r1?

address	data	instruction	r1	r2	r3	@100	@108
...	...	...	...	100	108	42	...
1000	e5921000	ldr r1, [r2]	42	100	108	42	...
1004	e5832000	str r1, [r3]	42	100	108	42	42
1008	e2811001	add r1, r1, #1					

A Worked Example

What does this code do? What ends up in r1?

address	data	instruction	r1	r2	r3	@100	@108
...	...	...	...	100	108	42	...
1000	e5921000	ldr r1, [r2]	42	100	108	42	...
1004	e5832000	str r1, [r3]	42	100	108	42	42
1008	e2811001	add r1, r1, #1	**43**	100	108	42	42

A Worked Example

What does this code do? What ends up in r1?

address	data	instruction	r1	r2	r3	@100	@108
...	...	...	...	100	108	42	...
1000	e5921000	ldr r1, [r2]	42	100	108	42	...
1004	e5832000	str r1, [r3]	42	100	108	42	42
1008	e2811001	add r1, r1, #1	**43**	100	108	42	42

Most code is like the above, and it's easy to understand;
The challenge here is how to express that formally.

A Worked Example

Another look at the example:

What value ends up in r1 now?

```
                                        r1      r2      r3      @1000     @1008
  ...      ...      ...                 ...     1000    1008    e5921000  ...
1000     e5921000   ldr r1, [r2]
1004     e5832000   str r1, [r3]
1008     e2811001   add r1, r1, #1
```

A Worked Example

Another look at the example:

What value ends up in r1 now?

			r1	r2	r3	@1000	@1008
...	...	...	...	1000	1008	e5921000	...
1000	e5921000	ldr r1, [r2]	e5921000	1000	1008	e5921000	...
1004	e5832000	str r1, [r3]					
1008	e2811001	add r1, r1, #1					

A Worked Example

Another look at the example:

What value ends up in r1 now?

			r1	r2	r3	@1000	@1008
...	...	...	...	1000	1008	e5921000	...
1000	e5921000	ldr r1, [r2]	e5921000	1000	1008	e5921000	...
1004	e5832000	str r1, [r3]	e5921000	1000	1008	e5921000	e5921000
1008	e2811001	add r1, r1, #1					

A Worked Example

Another look at the example:

What value ends up in r1 now?

			r1	r2	r3	@1000	@1008
...	...	...	...	1000	1008	e5921000	...
1000	e5921000	ldr r1, [r2]	e5921000	1000	1008	e5921000	...
1004	e5832000	str r1, [r3]	e5921000	1000	1008	e5921000	e5921000
1008	e2811001	add r1, r1, #1	**e5921001**	1000	1008	e5921000	e5921000

A Worked Example

Another look at the example:

What value ends up in r1 now?

			r1	r2	r3	@1000	@1008
...	...	...	...	1000	1008	e5921000	...
1000	e5921000	ldr r1, [r2]	e5921000	1000	1008	e5921000	...
1004	e5832000	str r1, [r3]	e5921000	1000	1008	e5921000	e5921000
1008	e2811001	add r1, r1, #1	**e5921001**	1000	1008	e5921000	e5921000

Wait a minute, what was that address? Didn't we just overwrite this instruction?

A Worked Example

Another look at the example:

What value ends up in r1 now?

			r1	r2	r3	@1000	@1008
...	...	...	...	1000	1008	e5921000	...
1000	e5921000	ldr r1, [r2]	e5921000	1000	1008	e5921000	...
1004	e5832000	str r1, [r3]	e5921000	1000	1008	e5921000	e5921000
1008	e2811001	add r1, r1, #1	**e5921001**	1000	1008	e5921000	e5921000

Wait a minute, what was that address? Didn't we just overwrite this instruction?

1008	e5921000	ldr r1, [r2]	**e5921000**	1000	1008	e5921000	e5921000

A Worked Example

Another look at the example:

What value ends up in r1 now?

			r1	r2	r3	@1000	@1008
...	...	...	...	1000	1008	e5921000	...
1000	e5921000	ldr r1, [r2]	e5921000	1000	1008	e5921000	...
1004	e5832000	str r1, [r3]	e5921000	1000	1008	e5921000	e5921000
1008	e2811001	add r1, r1, #1	**e5921001**	1000	1008	e5921000	e5921000

Wait a minute, what was that address? Didn't we just overwrite this instruction?

1008	e5921000	ldr r1, [r2]	**e5921000**	1000	1008	e5921000	e5921000

Which of these is the right answer?

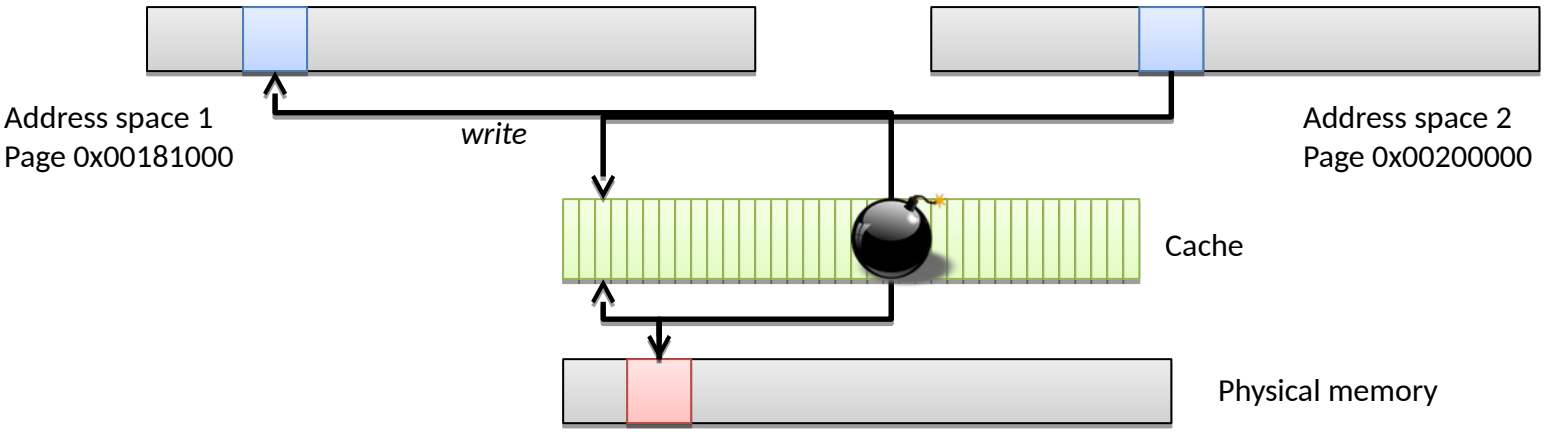
The ISA Isn't Enough

- The core ISA is pretty easy to model, and well documented.
  - ARMv6 was 1600 lines in Lyrebird.
- Interacting mechanisms are **hard**.
  - Heavily dependent on microarchitecture.
  - ***This is where the bugs sneak in.***
- For seL4 we went with a very simplistic HW model: "Surely the hardware can't be *that* weird?".

- Spoiler:
  - Both confinement and information flow proofs are undermined by exactly these sorts of bugs (details shortly).

Errata (Hardware Bugs)

Cache Bombs



- Unmap a frame from AS 1 with a dirty cache line
- Map the same frame into somewhere else (AS2)
- At some **unpredictable** time, the cache will write the line. ***BOOM!***

You Can't Trust the Hardware

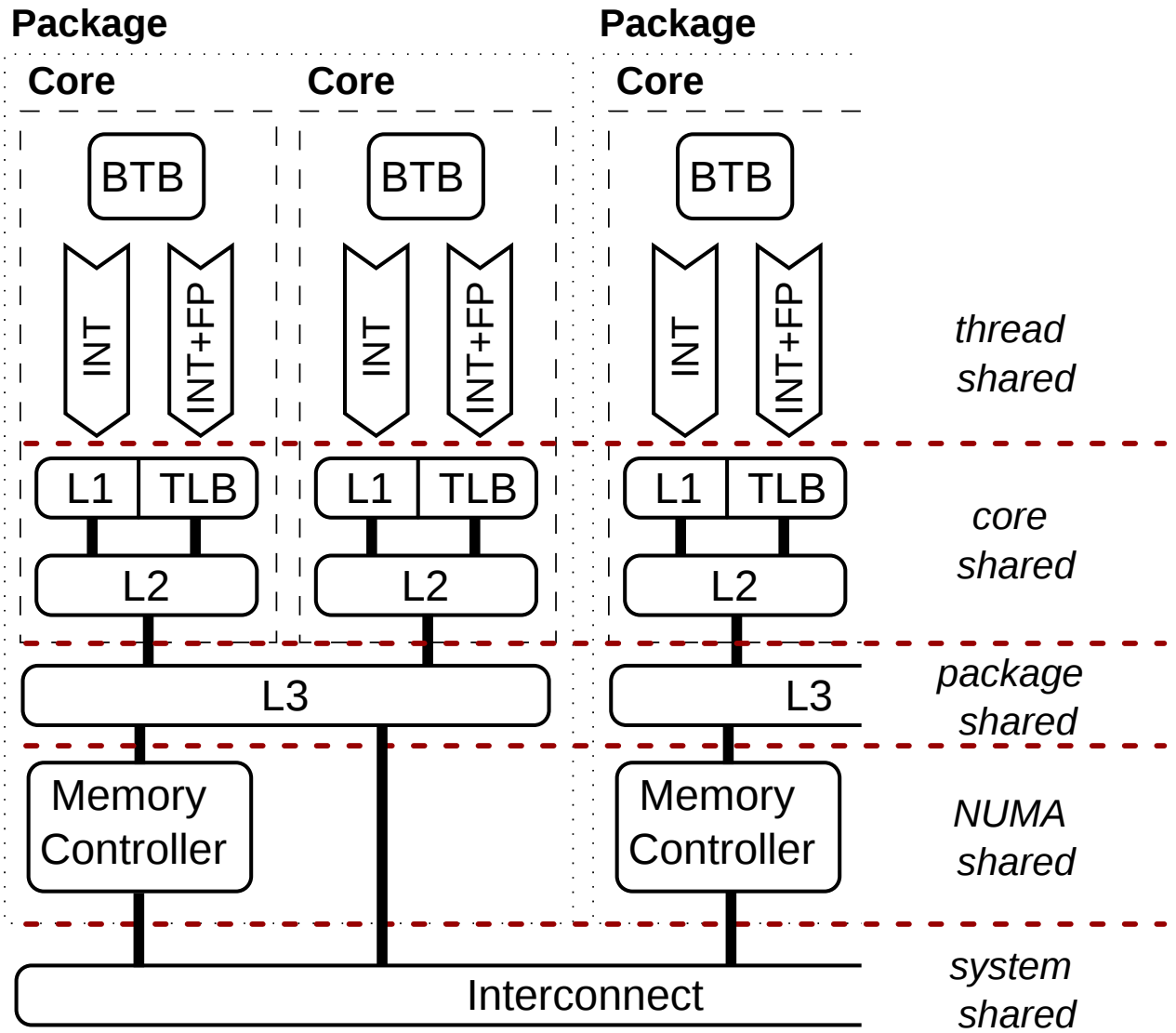
- seL4 was verified *modulo a hardware model*.
- The Cortex A8 has bugs:
  - Cache flushes don't work.
  - As of today, these “errata” are **still** not public.
  - We rediscovered these by accident.
- Non-coherent memory is coming.

Source: Chip Errata for the i.MX51, Freescale Semiconductor

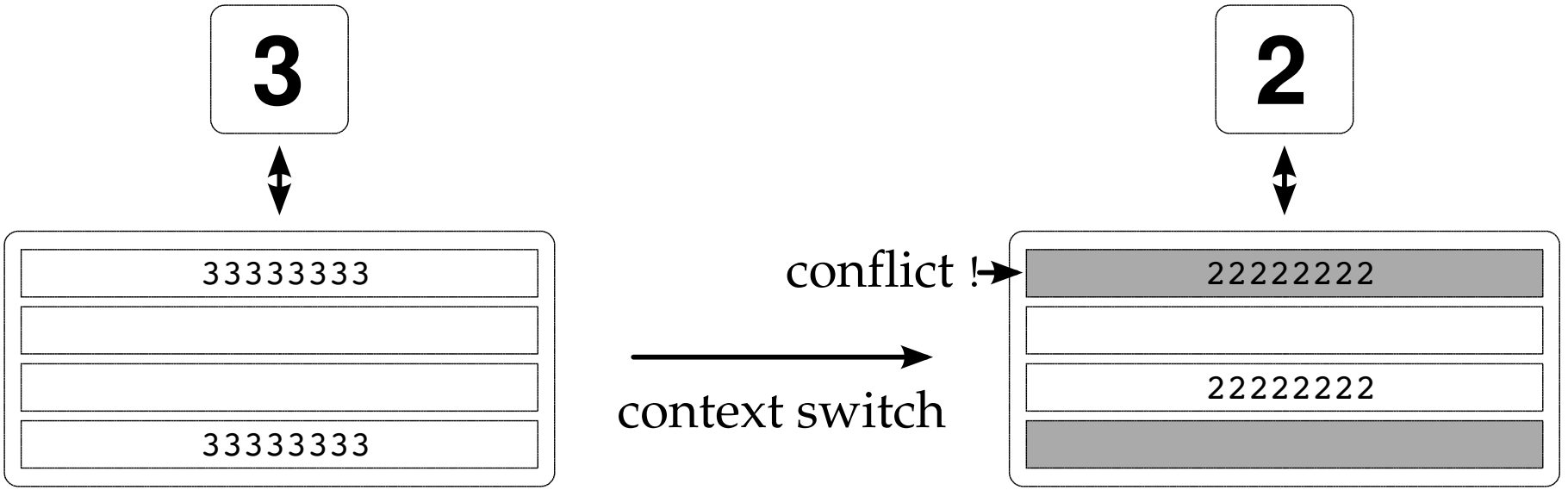
ENGcm09830	ARM: Load and Store operations on the shared device memory regions may not complete in program order	No fix scheduled	12
ENGcm07788	ARM: A RAW hazard on certain CP15 registers can result in a stale register read	No fix scheduled	14
ENGcm04786	ARM: ARPROT[0] is incorrectly set to indicate a USER transaction for memory accesses generated from user tablewalks	No fix scheduled	16
ENGcm04785	ARM: C15 Cache Selection Register (CSSELR) is not banked	No fix scheduled	18
ENGcm07784	ARM: Cache clean memory ops generated by the Preload Engine or Clean by MVA to PoC instructions may corrupt the memory	No fix scheduled	19
ENGcm07786	ARM: Under a specific set of conditions, a cache maintenance operation performed by MVA can result in memory corruption	No fix scheduled	21
ENGcm07782	ARM: Clean and Clean/Invalidate maintenance ops by MVA to PoC may not push data to external memory	No fix scheduled	23
ENGcm04758	ARM: Incorrect L2 cache eviction can occur when L2 is configured as an inner cache	No fix scheduled	25
ENGcm04761	ARM: Swap instruction, preload instruction, and instruction fetch request can interact and cause deadlock	No fix scheduled	26
ENGcm04759	ARM: NEON load data can be incorrectly forwarded to a subsequent request	No fix scheduled	28
ENGcm04760	ARM: Under a specific set of conditions, processor deadlock can occur when L2 cache is servicing write allocate memory	No fix scheduled	30
ENGcm10230	ARM: Clarification regarding the ALP bits in AMC register	No fix scheduled -Clarified in RM	32
ENGcm10700	ARM: If a Perf Counter OVFL occurs simultaneously with an update to a CP14 or CP15 register, the OVFL status can be lost	No fix scheduled	33
ENGcm10716	ARM: A Neon store to device memory can result in dropping a previous store	No fix scheduled	35
ENGcm10701	ARM: BTB invalidate by MVA operations do not work as intended when the IBE bit is enabled	No fix scheduled	37
ENGcm10703	ARM: Taking a watchpoint is incorrectly prioritized over a precise data abort if both occur simultaneously on the same address	No fix scheduled	39
ENGcm10724	ARM: VCVT.f32.u32 can return wrong result for the input 0xFFFF_FF01 in one specific configuration of the floating point unit	No fix scheduled	41

Side Channels

Resource Sharing in Modern CPUs



The Cache Contention Channel

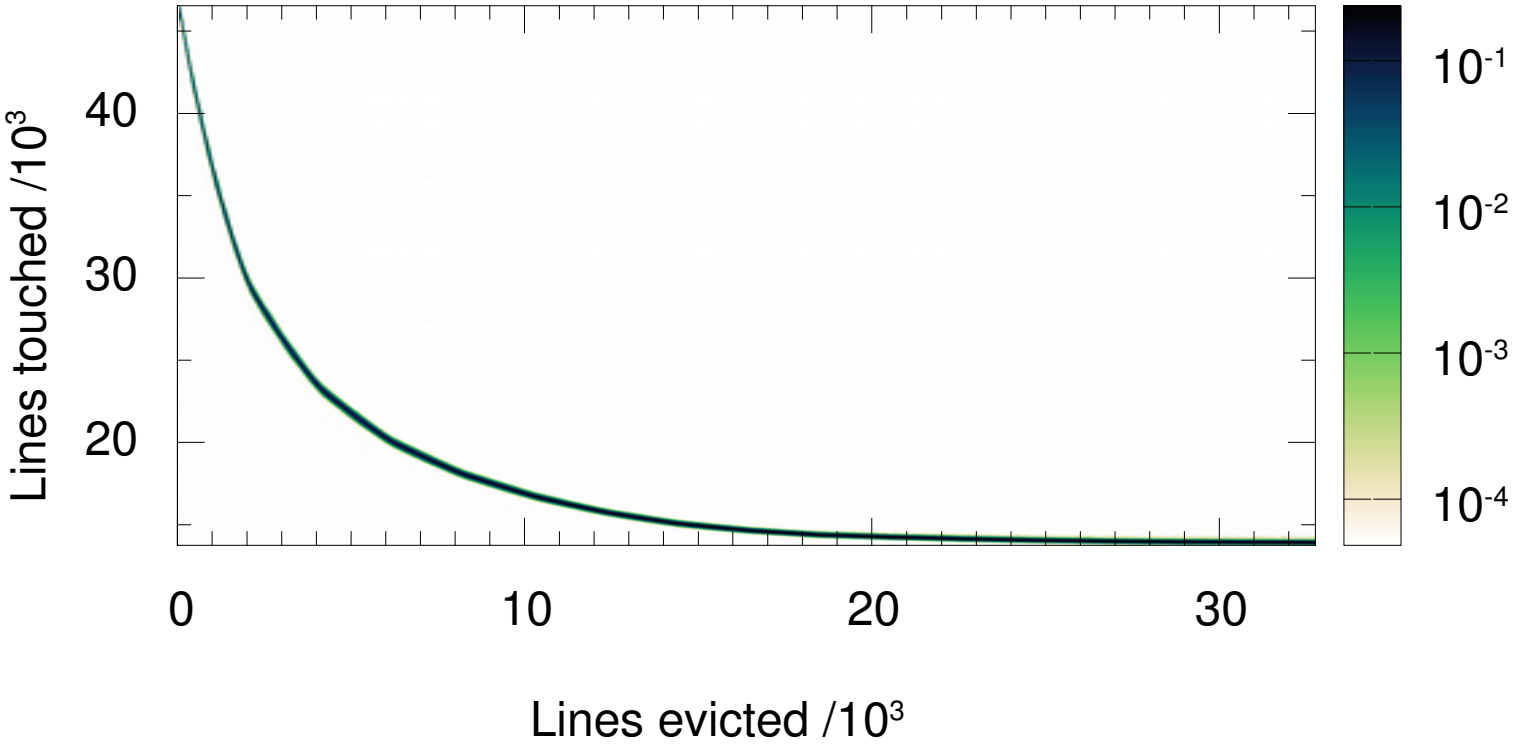


Empirical Evaluation on seL4

	Core	Date	L2 Cache
iMX.31	ARM1136JF-S (*ARMv6*)	2005	128 KiB
E6550	Conroe (*x86-64*)	2007	4096 KiB
DM3730	Cortex A8 (*ARMv7*)	2010	256 KiB
AM3358	Cortex A8 (*ARMv7*)	2011	256 KiB
iMX.6	Cortex A9 (*ARMv7*)	2011	1024 KiB
Exynos4412	Cortex A9 (*ARMv7*)	2012	1024 KiB

- 7 years and 3 (ARM) core generations.
- 32-fold range of cache sizes.

Exynos4412 Cache Channel



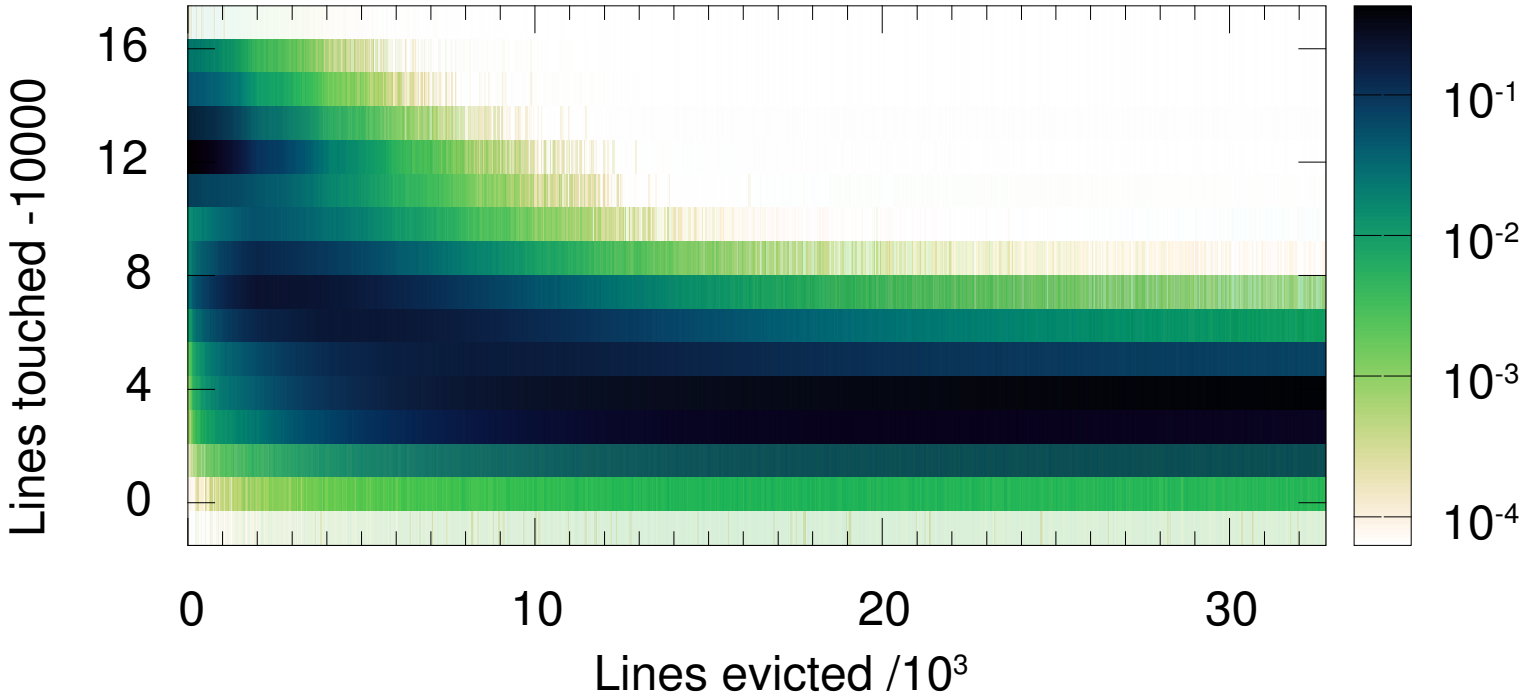
- 32,768 cache lines, 1000Hz sample rate (preemption).
- Bandwidth: 2400b/s.
- Baseline for comparison.

Instruction-Based Scheduling

The channel needs a clock. Tie it to progress, and the channel should vanish. This is a form of deterministic execution.

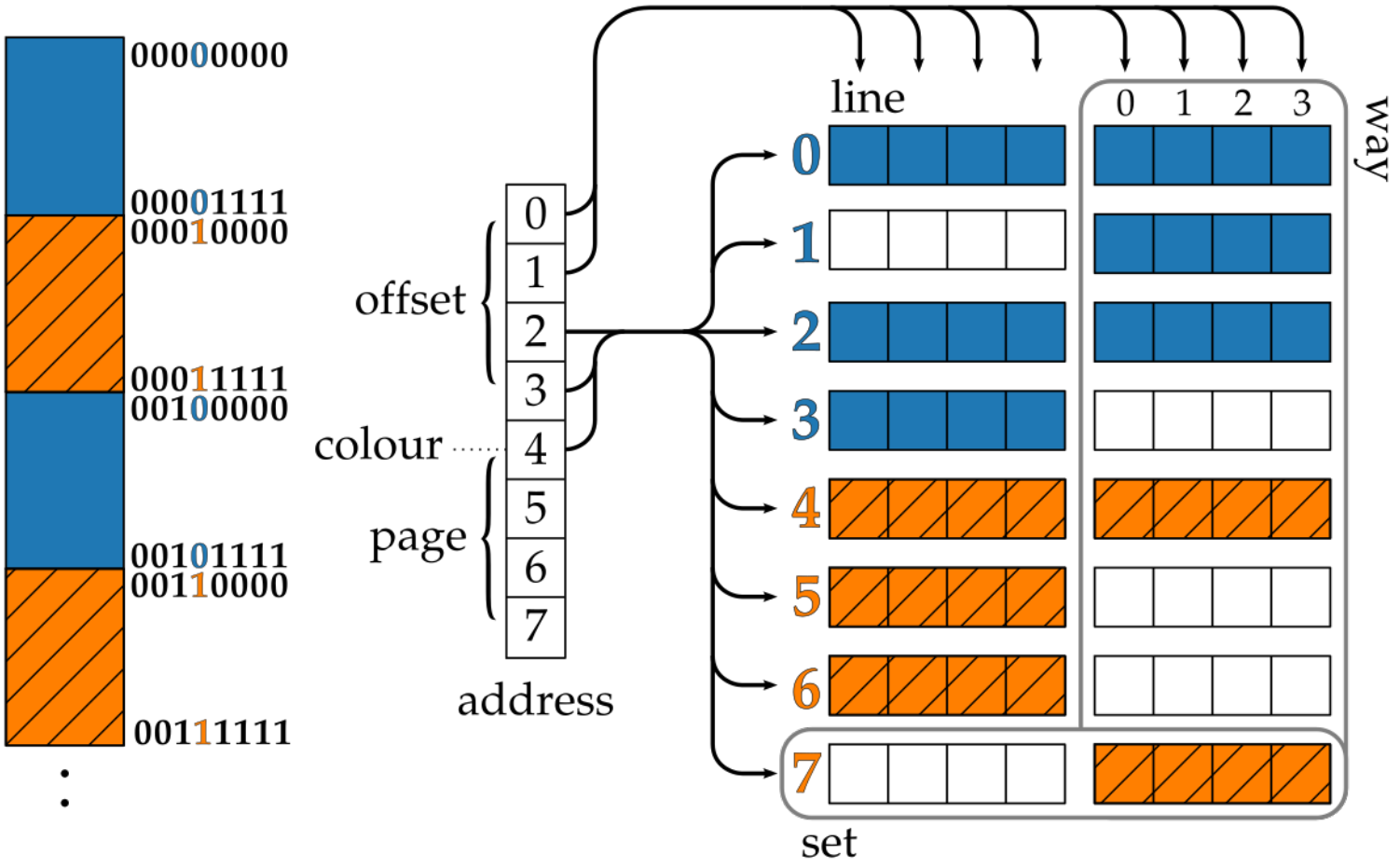
- Advantages:
  - Applies to any channel.
  - Simple to implement (18 lines in seL4).
- Disadvantages
  - Restrictive — Need to remove all clocks.
  - Performance counter accuracy critical.

Exynos4412 Cache Channel with IBS

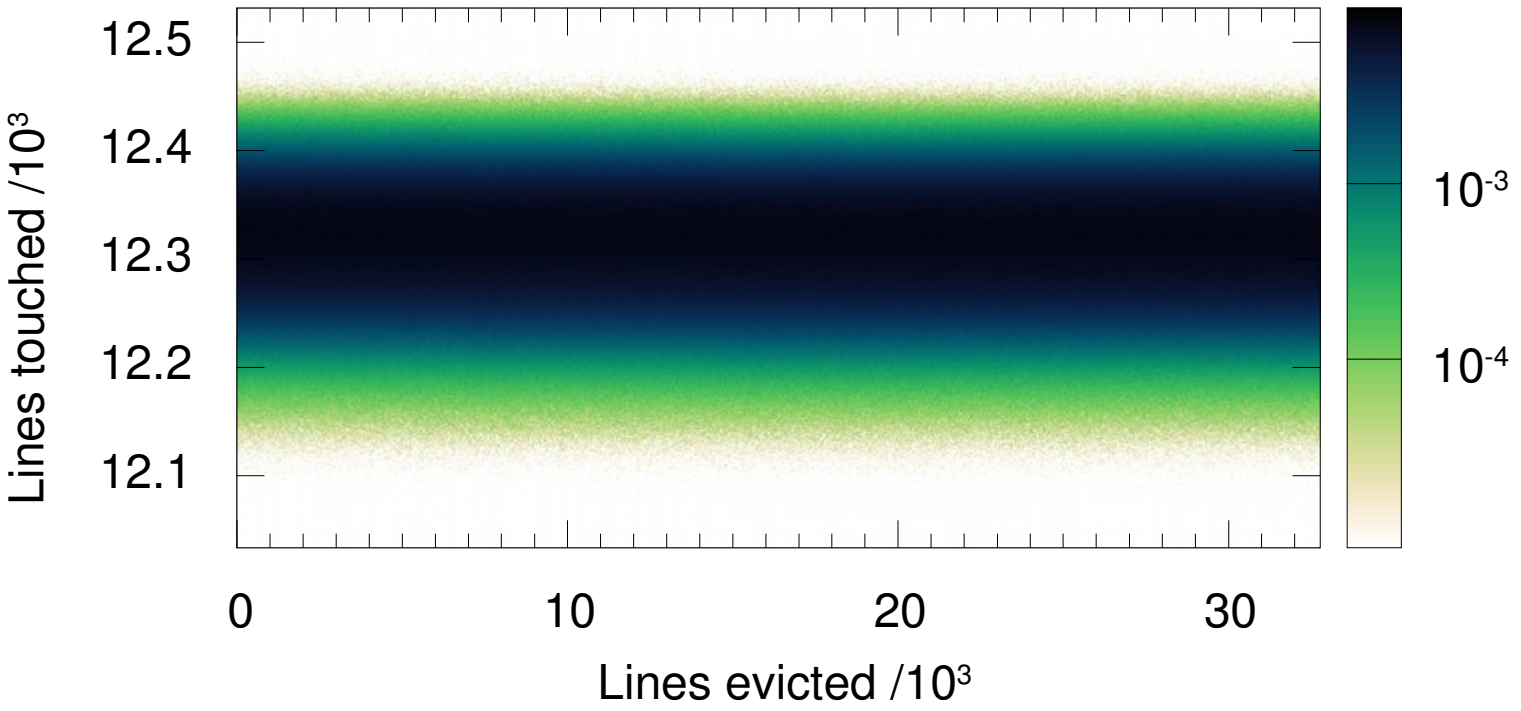


- Preempt after 10^5 instructions. Bandwidth 400b/s.
- Event delivery is imprecise thanks to speculation.

Cache Colouring

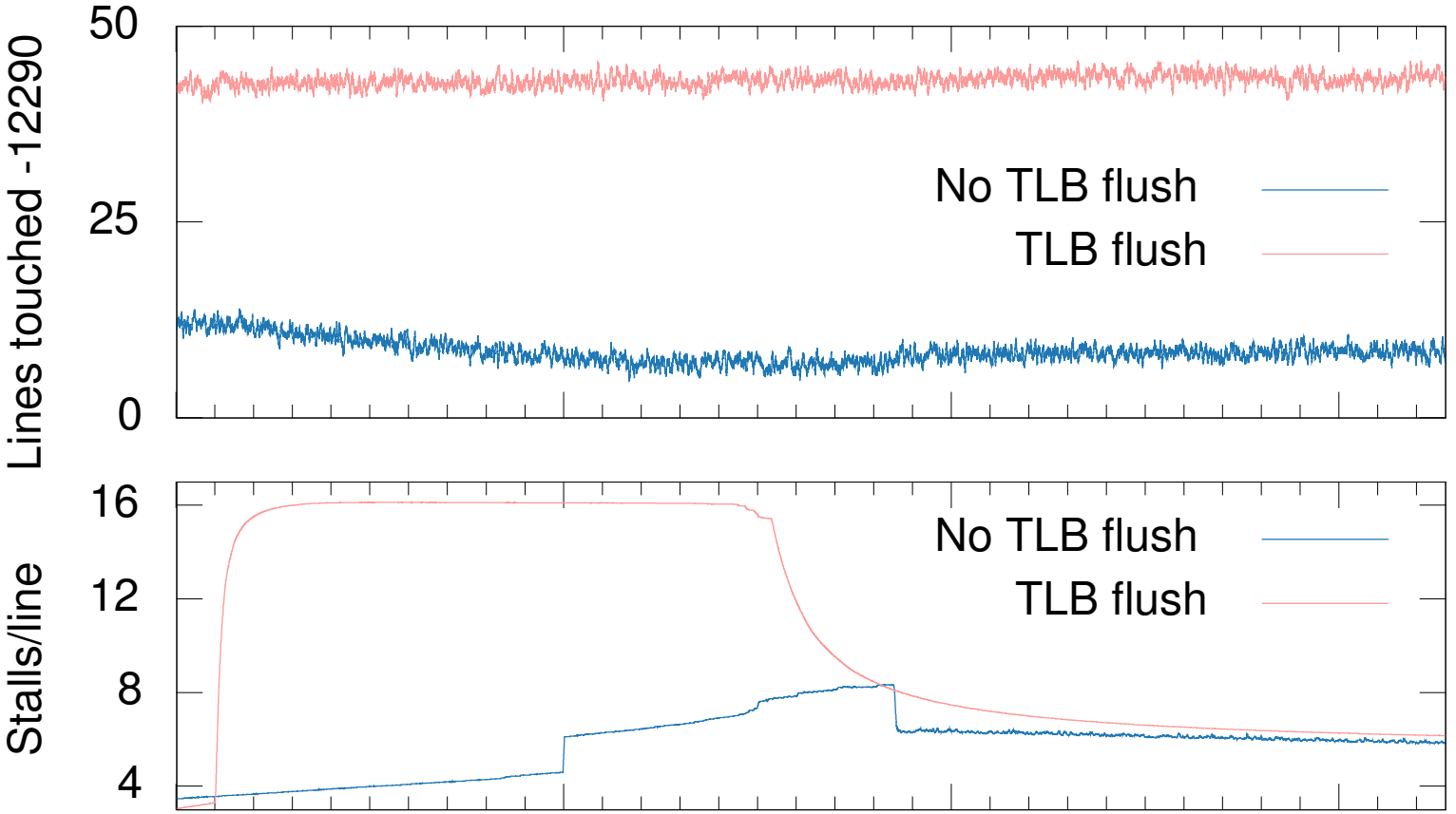


Exynos4412 Cache Channel, with Colouring

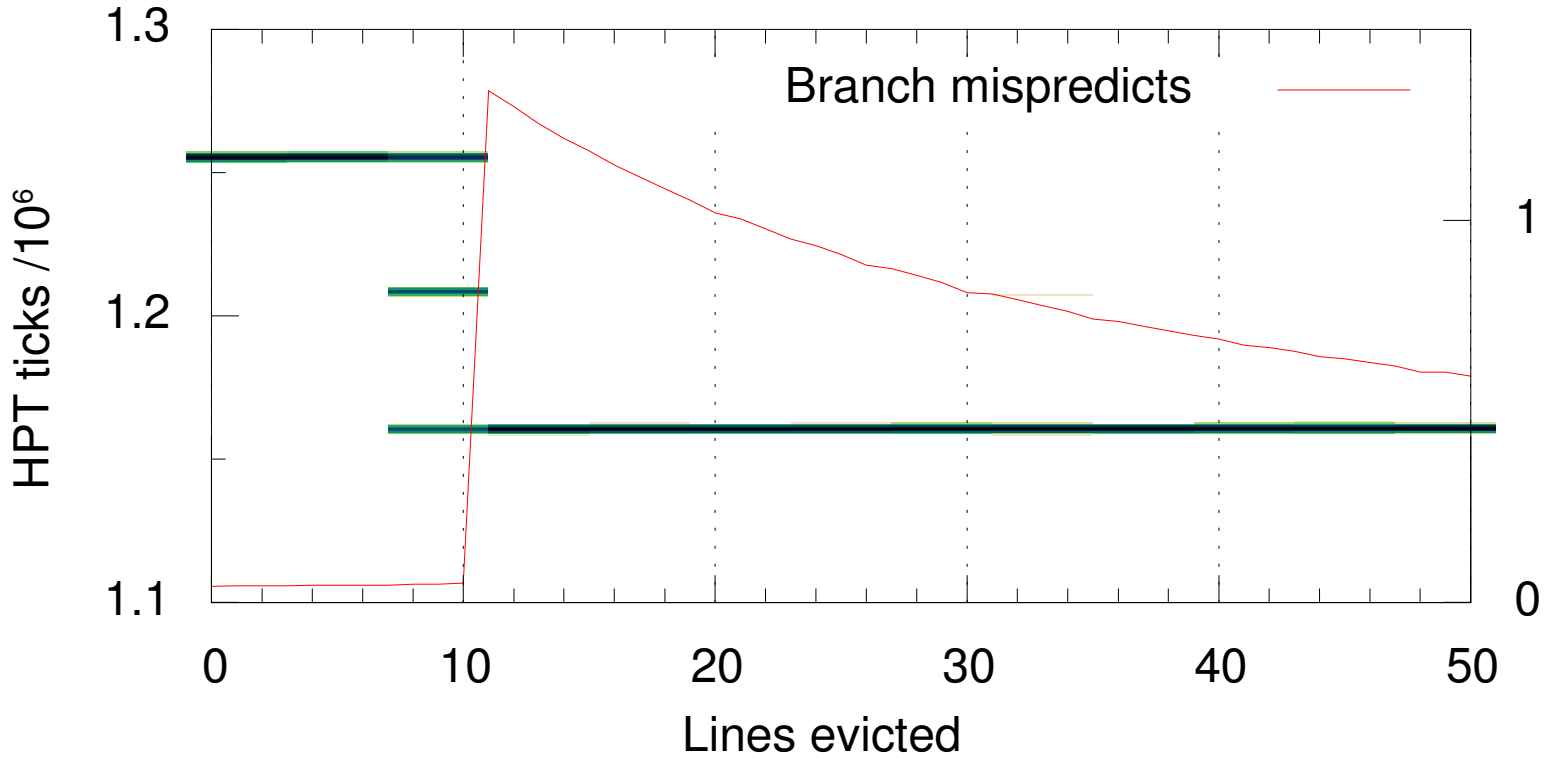


- Bandwidth: 15b/s. Where's that from?

Exynos4412 TLB Channel



Misprediction and the Cycle Counter



- Cycle counter affected by invisible mispredicts.
- A new (and **unexpected**) channel.
- Event delivery is **precise**, the cycle counter is **wrong**.

Summary so Far

- There are no trustworthy hardware models.
- The things our models hide *do* break security.

- There's some hope:
  - Formal ISA models exist (ARMv8 XML), but don't cover this stuff.
  - Hardware partitioning works, but still isn't well-enough specified.

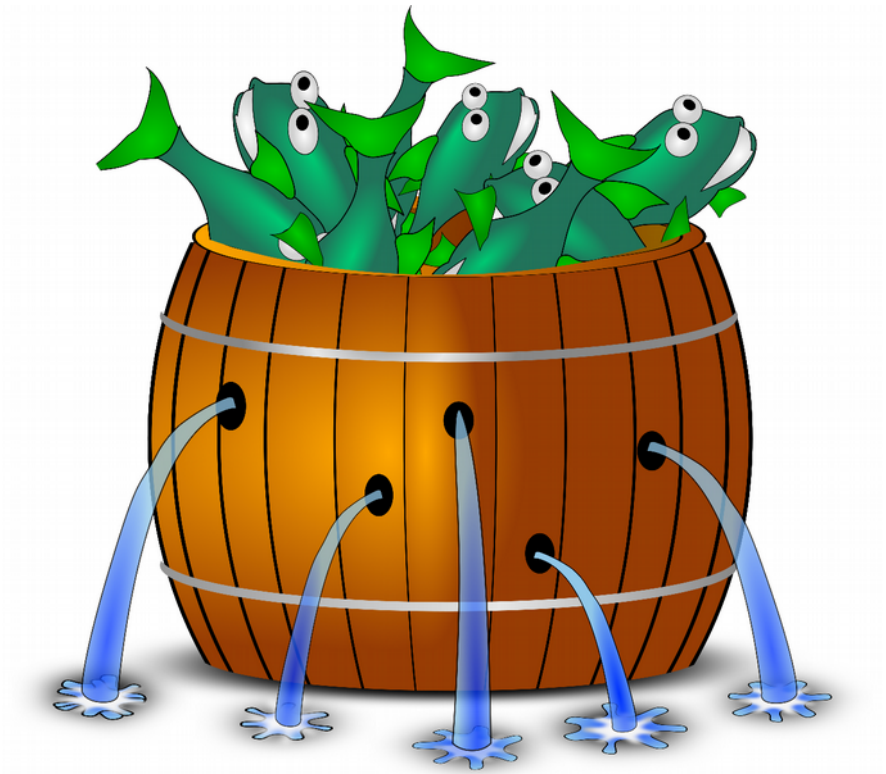
So, What Are We Doing About It?

1) Modelling hardware

2) Testing our models

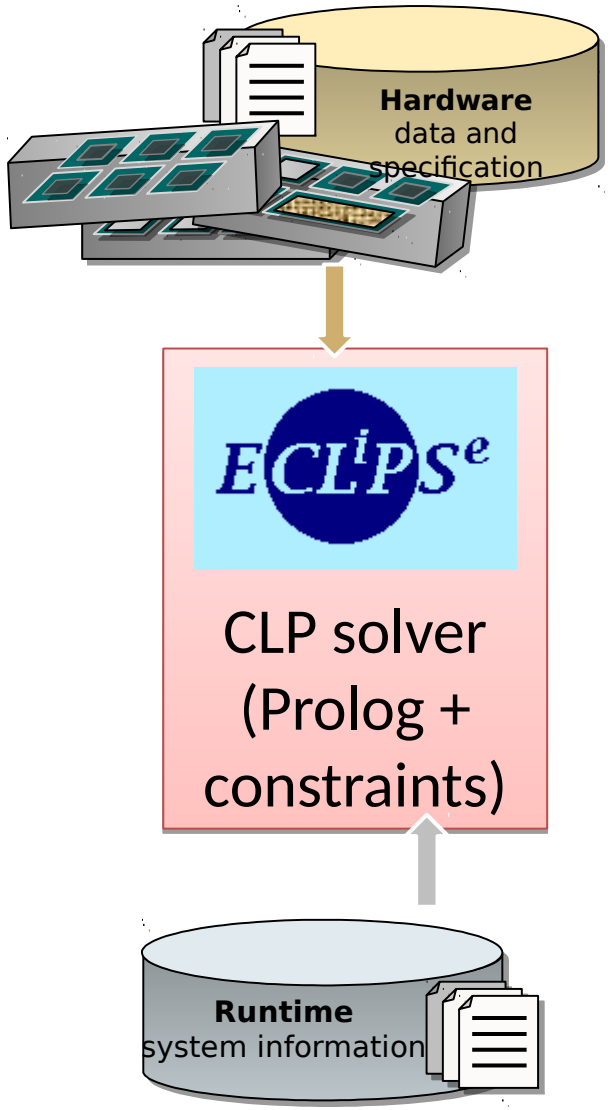
3) Building understandable hardware

Barrelfish



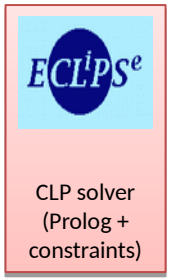
- seL4-related research OS
- Targets modern hardware (esp. multicore)
- Focus on automatic configuration and DSLs
- Info/Exo-kernel influence

The SKB



- System Knowledge Base
  - Hardware info
  - Runtime state
- Rich semantic model
  - Represent the hardware
  - Reason about it
  - Embed policy choices

What goes in?



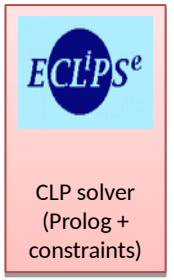
- Hardware resource discovery
  - E.g. PCI enumeration, ACPI, CPUID…
- Online hardware profiling
  - Inter-core all-pairs latency, cache measurements…
- Operating system state
  - Locks, process placement, etc.
- “Things we just know”
  - SoC specs, assertions from data sheets, etc.

Current SKB applications

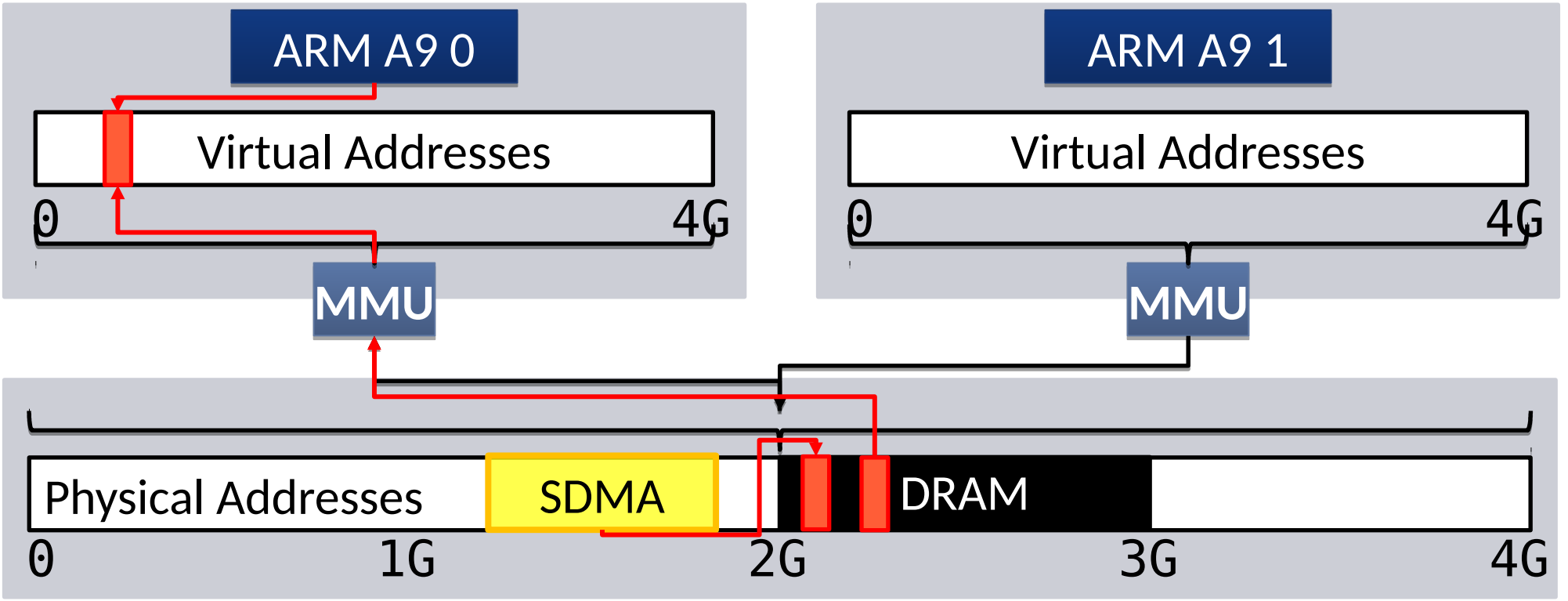
- General name server / service registry
- Coordination service / lock manager
- Device management
  - Driver startup / hotplug
- PCIe bridge configuration
  - A surprisingly hard CSAT problem!
- Intra-machine routing
  - Efficient multicast tree construction
- Cache-aware thread placement
  - Used by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

And now:

- Teach the SKB about microarchitecture!



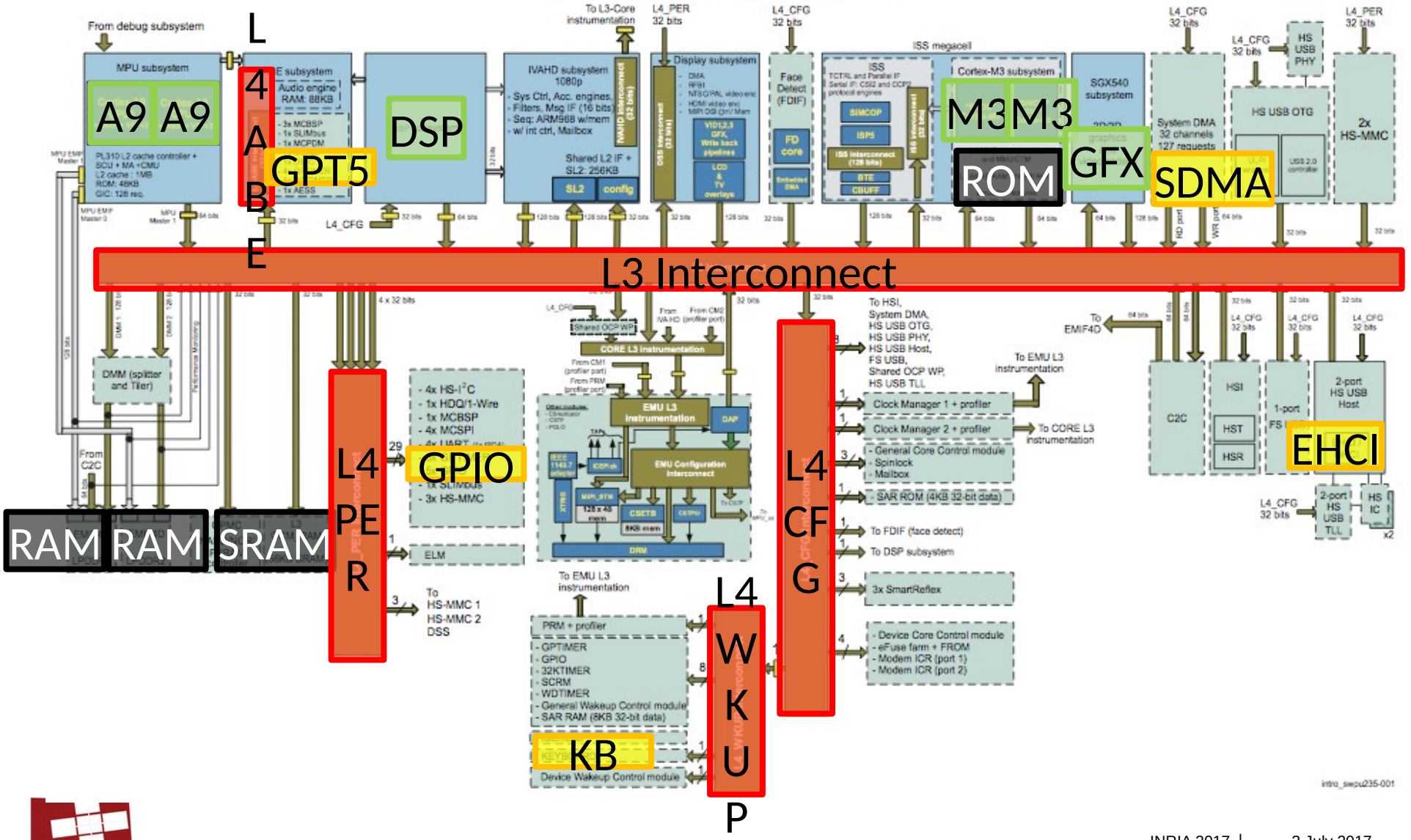
How I Picture a Computer



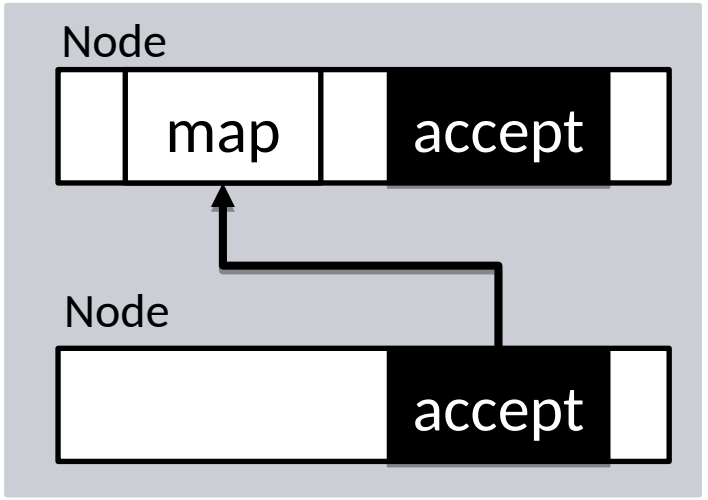
Ti OMAP 4460 SoC

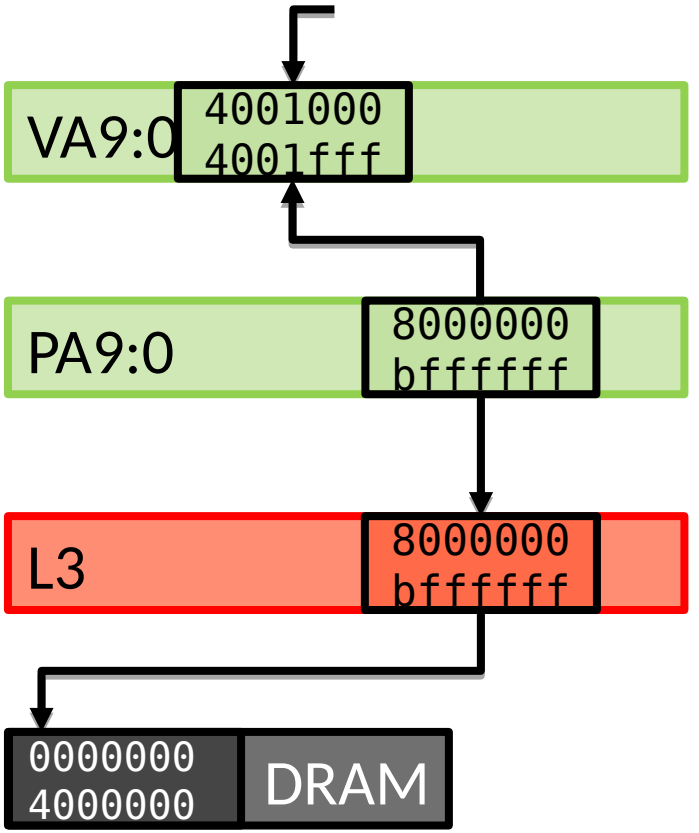
How the Computer *Actually* Looks

Your mobile phone… 5-10 years ago!

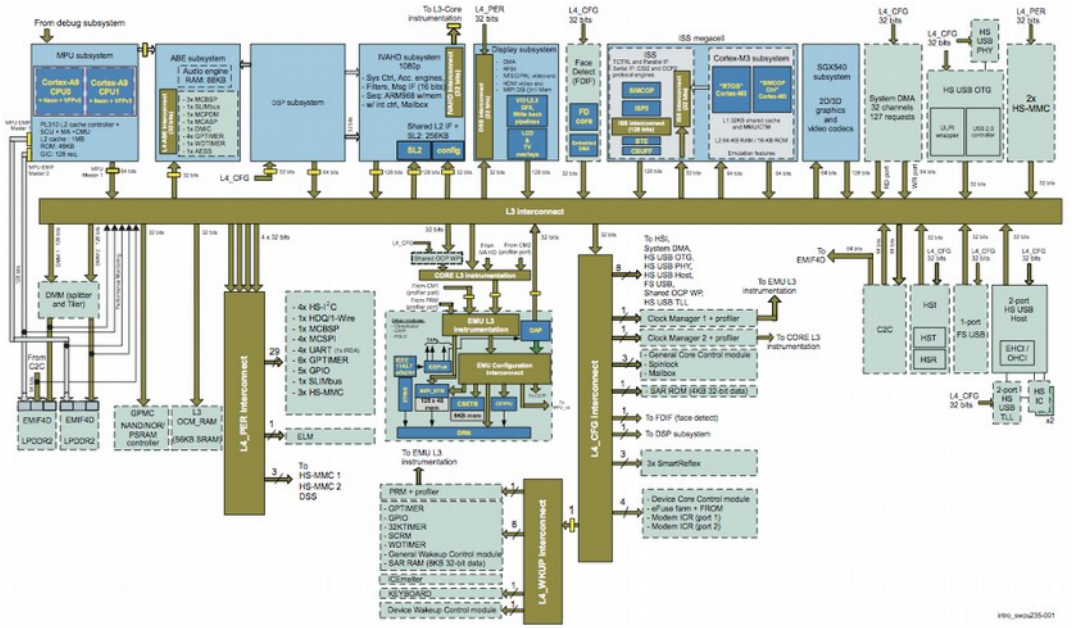


Decoding Nets





The OMAP4460 Decoding Net



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

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

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

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

P_{DSP} **is map** $[1d3e_3/12$ **to** GPT **at** $0]$ **over** $L3$

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

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

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

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

$L4$ **is map** $[49038_3/12$ **to** GPT **at** $0]$

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

GPT **is accept** $[0/12]$

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

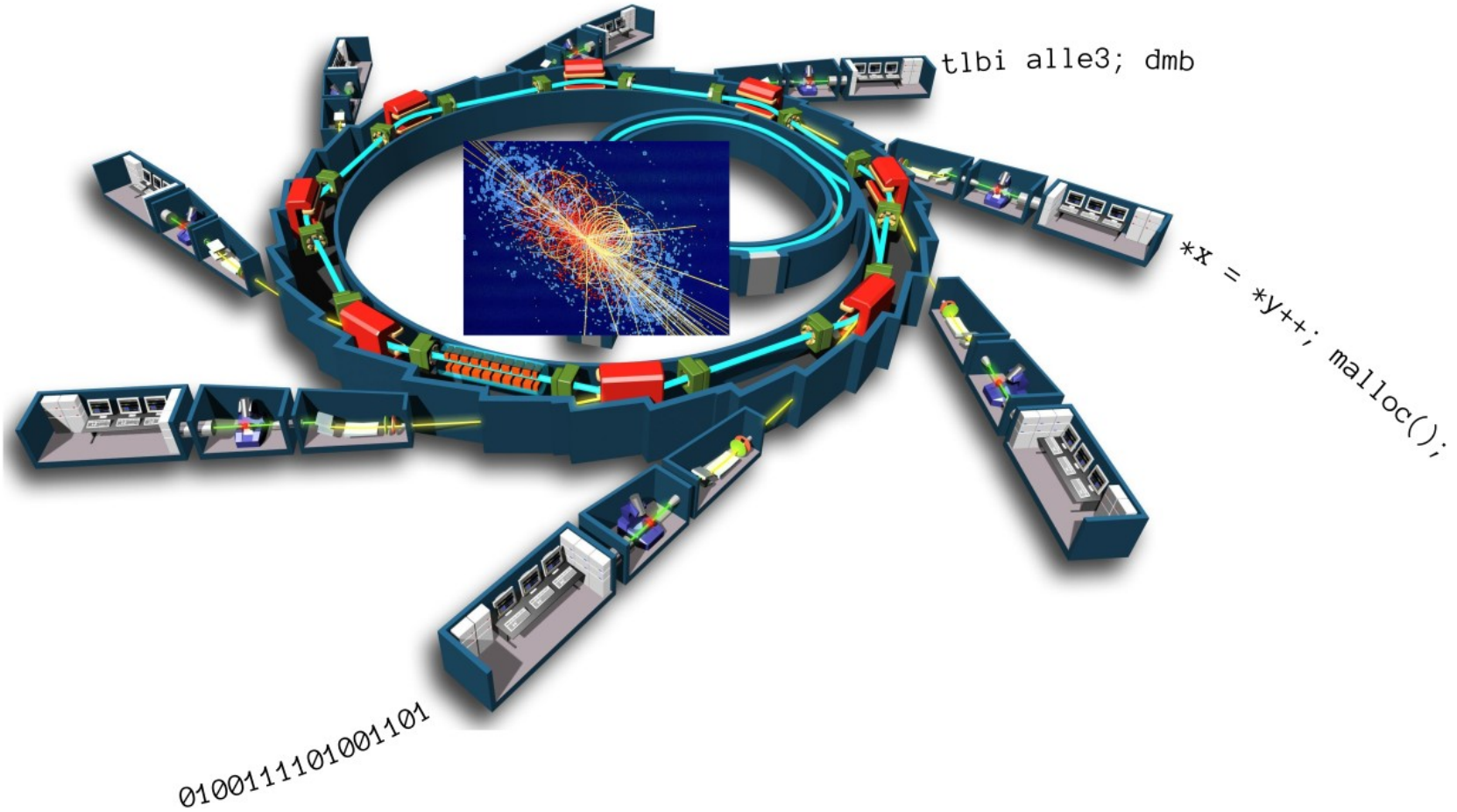
$L3$ **is map** $[49000_3/24$ **to** $L4$ **at** $40100_3, 55000_3/12$ **to** $MIF]$ **accept** $[80000_3/30]$

Using the model

- Static Configuration:
  - We can now generate the kernel page tables directly from the formal spec.
- Dynamic Discovery and Reconfiguration:
  - The SKB can be populated at runtime – extend the model as hardware is discovered.
- Scheduling:
  - We collaborate with the DB research group on operator scheduling – this work needs the model data.

Testing the Model

We're Building a *Large Program Collider*



Images: CERN; Chaix & Morel et associés

Collide *instructions* at 0.99*c*, and observe the decay products.

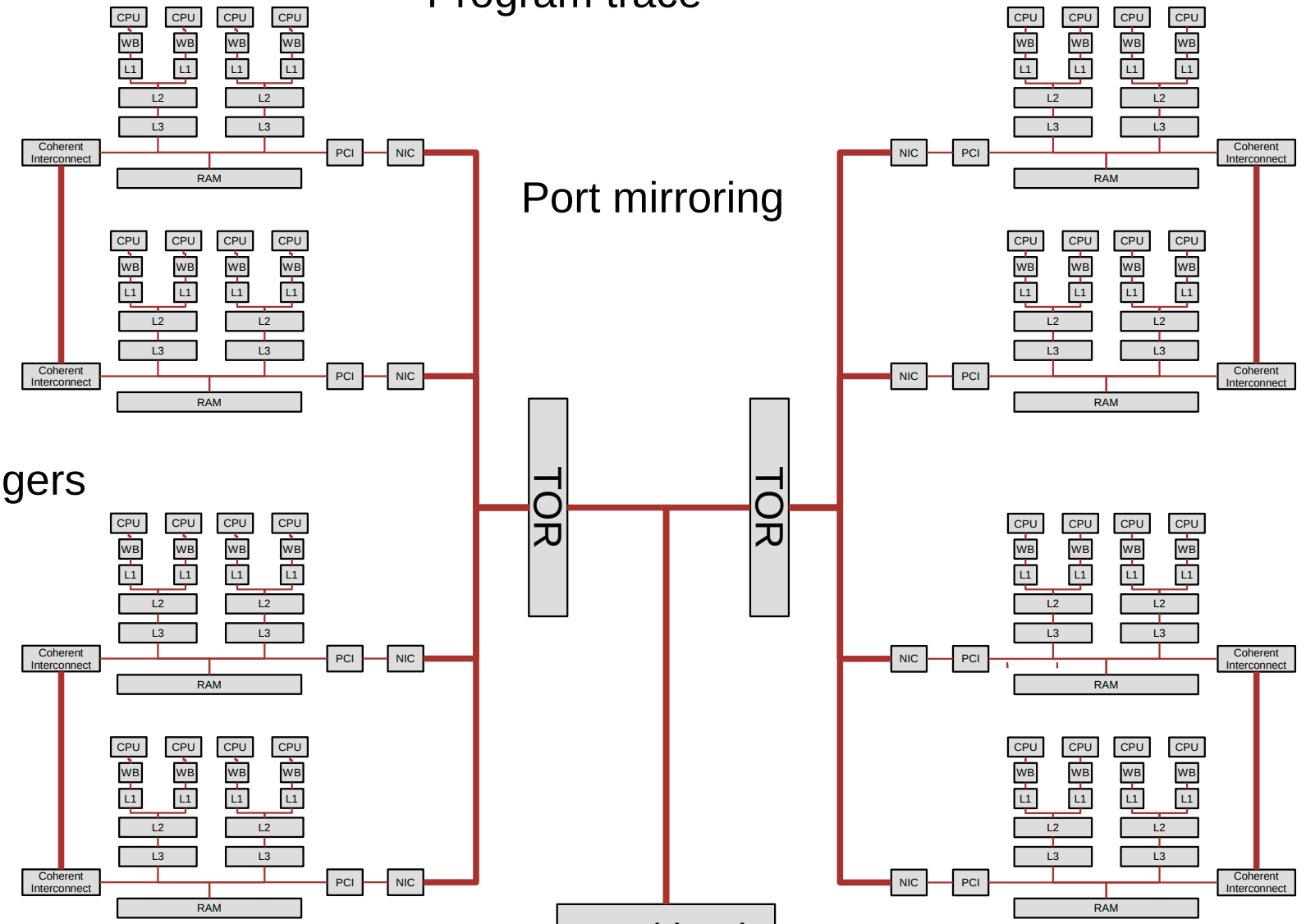
There's a Lot of Data Available

Cache dumps

Program trace

Port mirroring

Event triggers

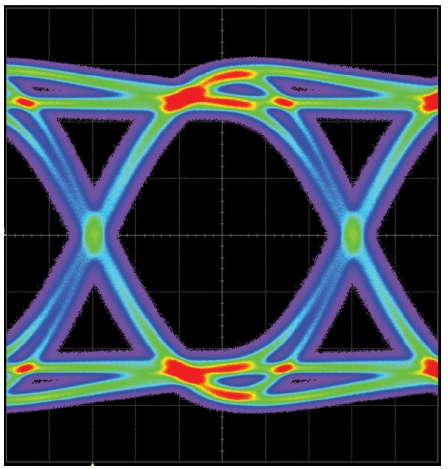


Openflow

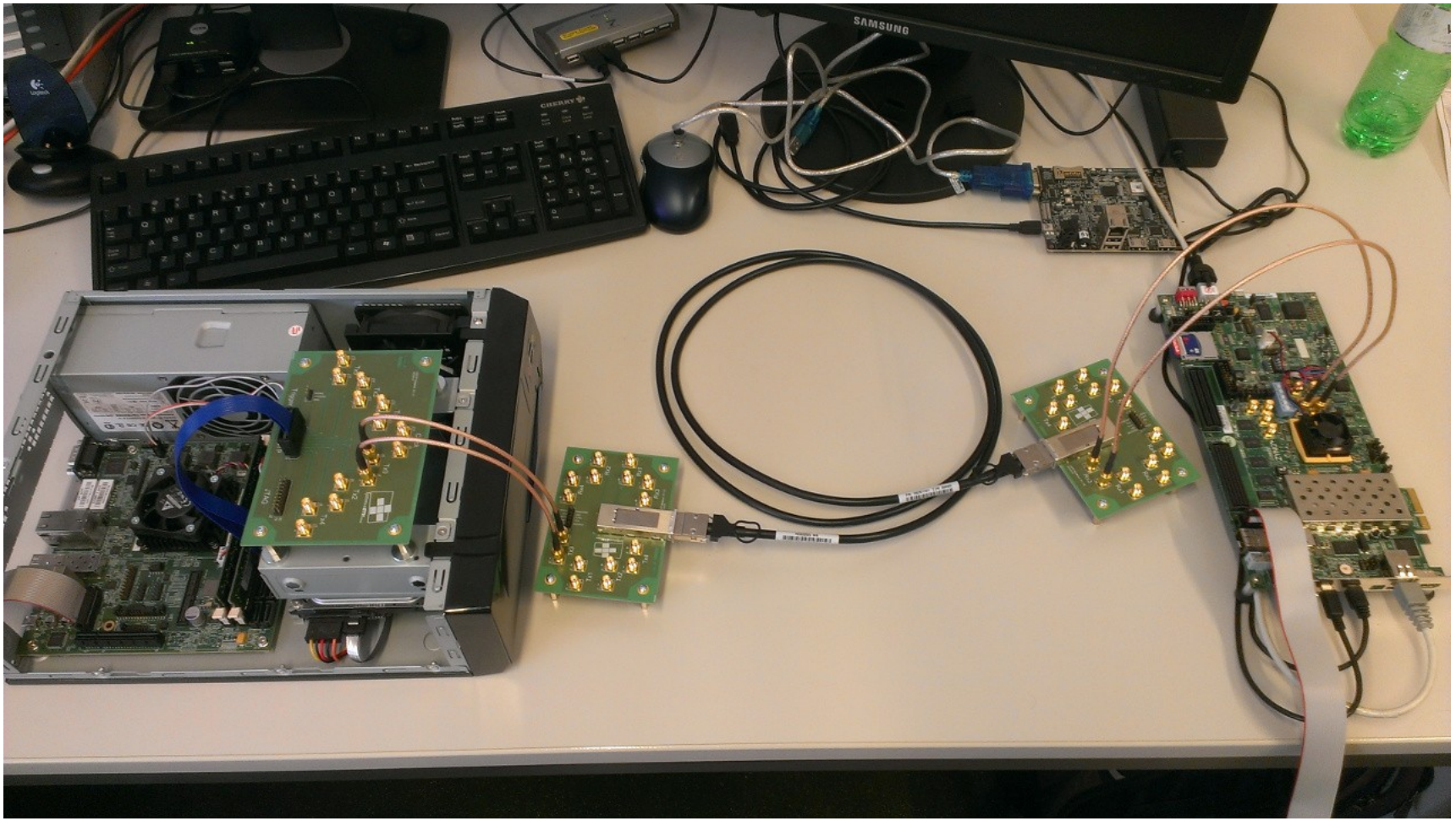
ARM High-Speed Serial Trace Port

- Streams from the *Embedded Trace Macrocell*.
- Cycle-accurate control flow + events @ 6GiB/s+
- Compatible with FPGA PHYs.
- Well-documented protocol.
  - Aurora 8/10
- Available on ARMv8

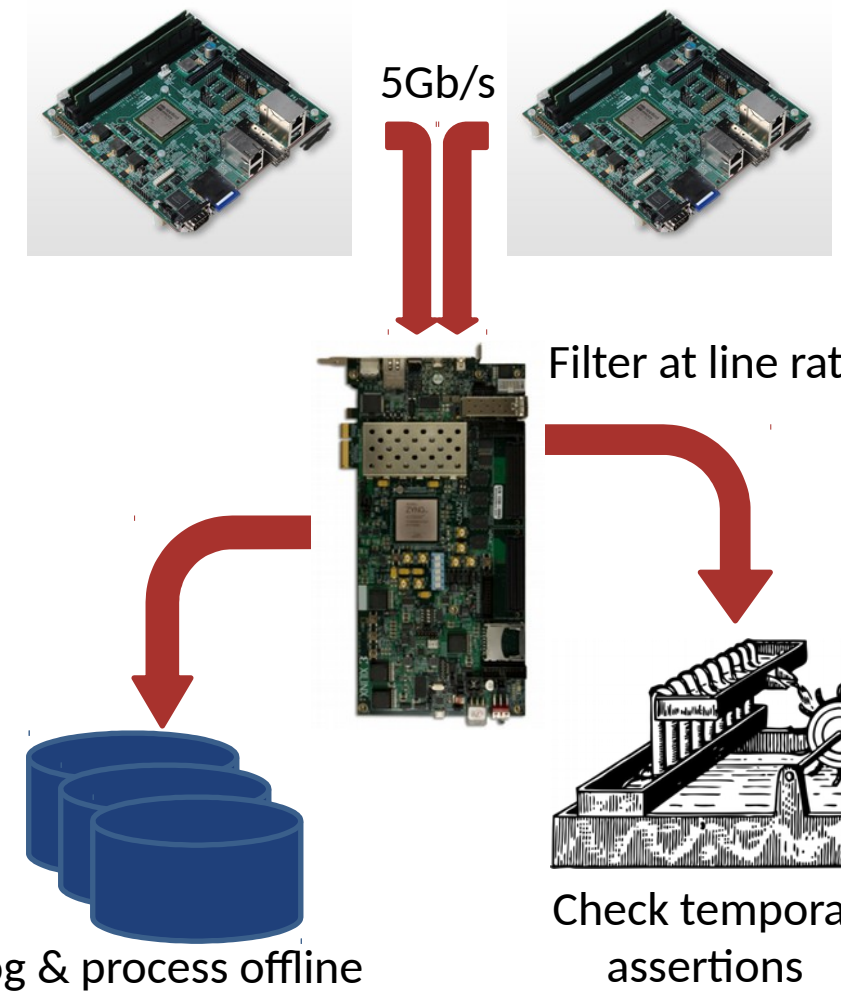
Image: Teledyne Lecroy



HSSTP Testbench



Hardware Tracing for Correctness



Are HW operations right?

$\exists va.va \rightarrow pa$

```
unmap(pa);
cleanDCache();
flushTLB();
```

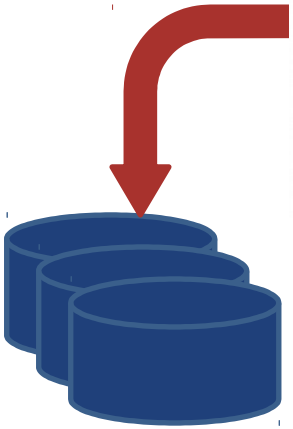
$\nexists va.va \rightarrow pa$

- Real time pipeline trace on ARM.
- Can halt and inspect caches.
- HW has “errata” (bugs).
- Check that it actually works!
- Catch transient and race bugs.

Hardware Tracing for Performance

5Gb/s

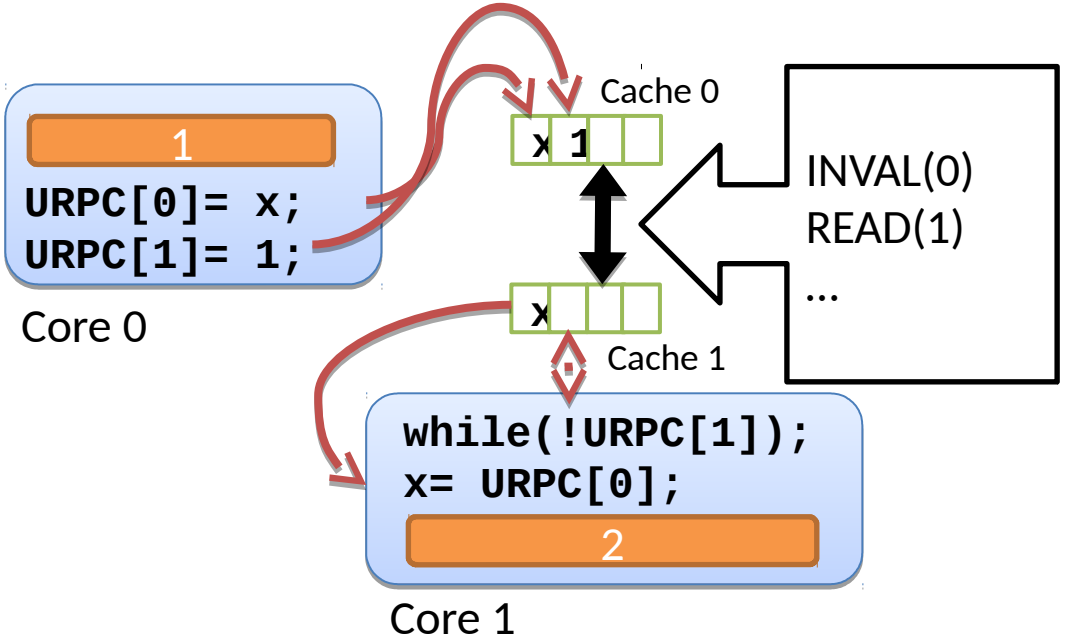
Filter at line rate



Log & process offline

- Should see N coherency messages.
- Do we?
  - The HW knows!

Is URPC optimal?



Properties to Check: Security

- Runtime verification is an established field.
- Lots of existing work to build on.
- What properties could we check efficiently?
- How could we map them to the filtering pipeline?


```
/* A very simple TESLA assertion. */
TESLA_WITHIN(example_syscall,
   previously(security_check(ANY(ptr),
                o, op) == 0));
```

http://www.cl.cam.ac.uk/research/security/ctsrd/tesla/

Properties to Check: Memory Management

```
void *a = malloc();
...
{a is still allocated}
free(a);
```

- Could we check this?

```
Gp $free(x) -> P !$free(x) S x = $malloc;
```

It's **always** been true that...

...if *x* is freed **now**, then...

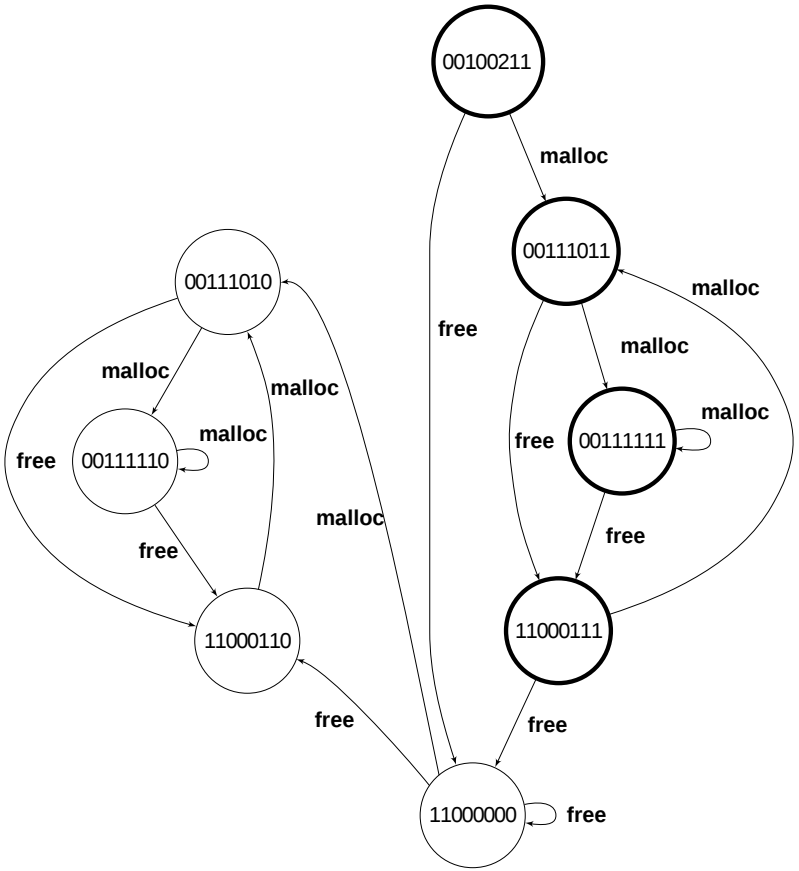
...**before** this free...

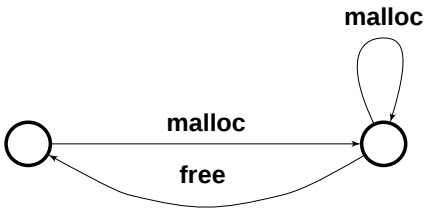
...there were no frees of *x*, **since** it was allocated.

Checking LTL with Automata

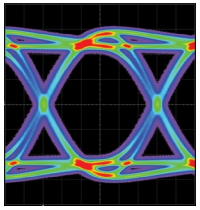
This is a well-studied problem, and standard algorithms exist:

```
Gp $free(x) -> P !$free(x) S x = $malloc;
```





A Streaming Verification Engine



Sources

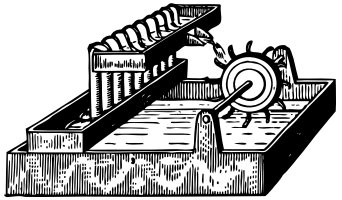
HSSTP

Packet Capture

Capture

ETM Sequencer

FPGA Capture



Processing

Dataflow Engine

FPGA Offload

Properties

TESLA

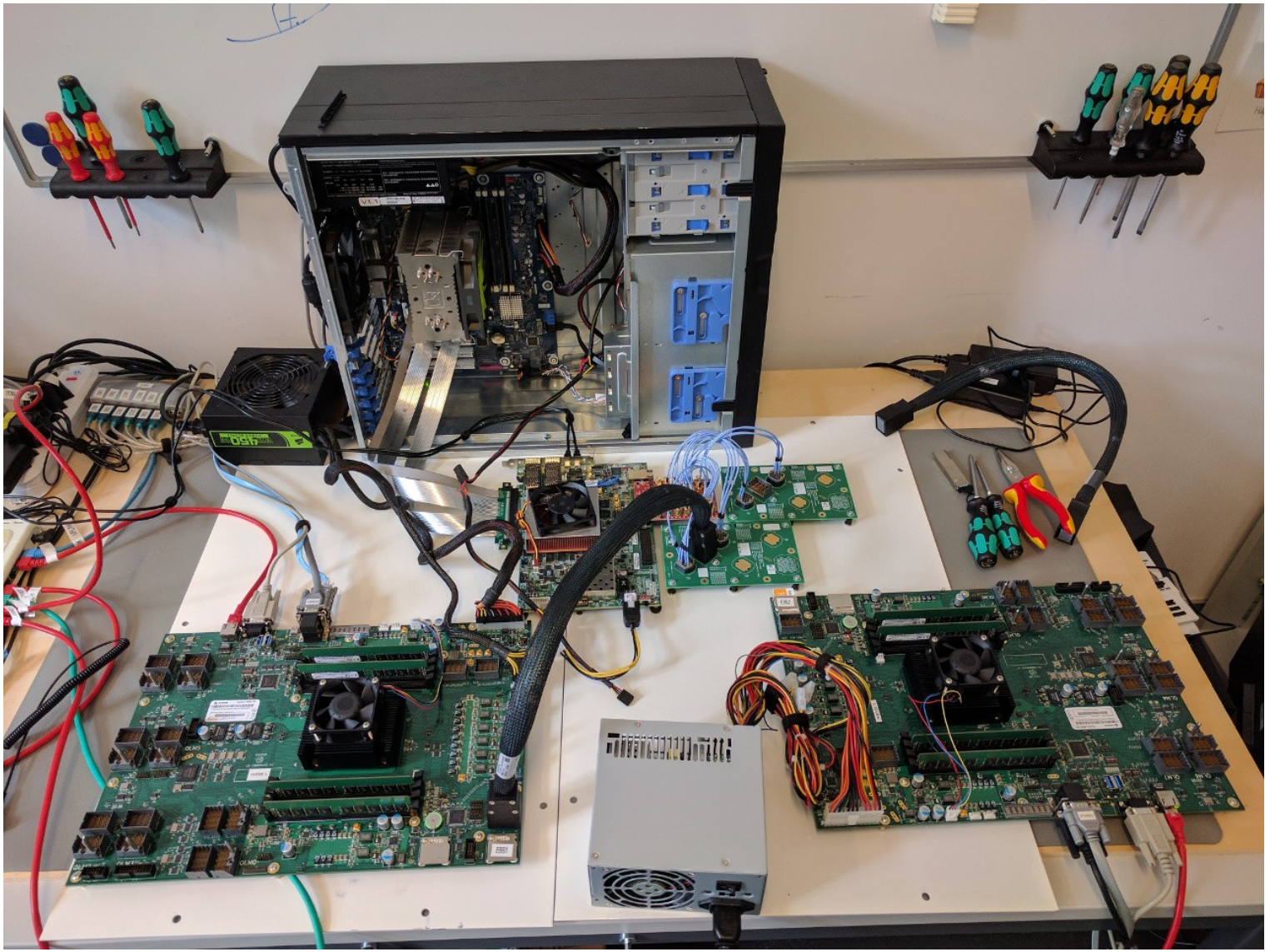
`malloc()` pairing

Coherence correctness

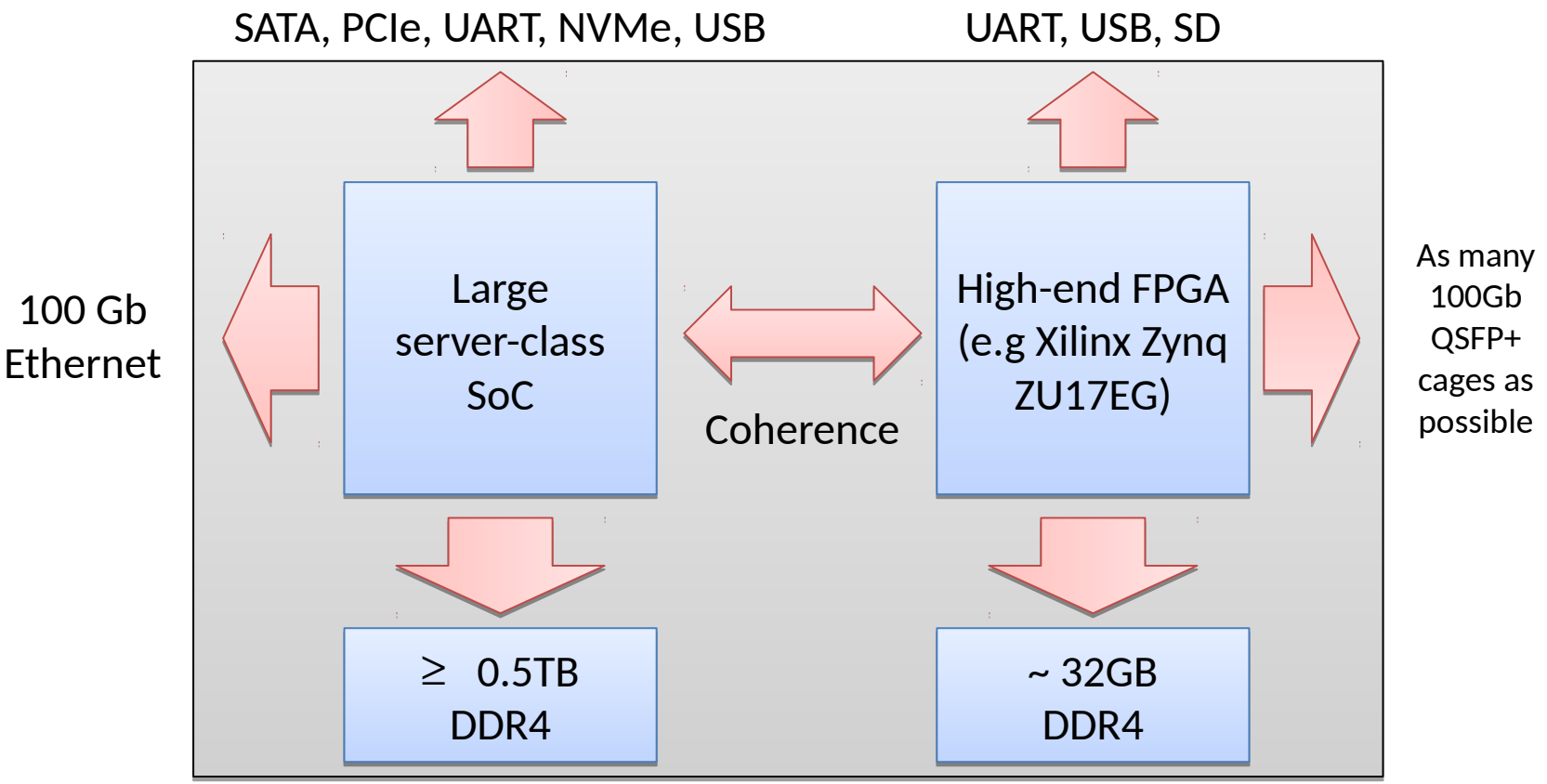
Constraints

Requirements

Building Understandable Hardware



Sketch


SATA, PCIe, UART, NVMe, USB
UART, USB, SD
100 Gb Ethernet
Large server-class SoC
Coherence
High-end FPGA (e.g Xilinx Zynq ZU17EG)
As many 100Gb QSFP+ cages as possible
≥ 0.5TB DDR4
~ 32GB DDR4