Hardware trace for system visibility

(again)

Al Grant
al.grant@arm.com

BSDCam 2017

The visibility problem

- What's going on?
- Kernel activity – e.g. performance-critical paths
- Userspace activity
- Interrupts
- Power management: DVFS, throttling, hotplug
- Microarchitectural events: cache, prefetch…
- Use of system resources: memory, power

Software trace

- programmer-inserted (printf debugging)
  - problem: no standard language mechanism
- compiler-inserted
  - e.g. `gcc -pg`
- DBT: DynInst, Pin, DynamoRIO: Dynamic binary instrumentation
  - DynamoRIO: https://github.com/DynamoRIO/dynamorio
  - ARMv8 support added 2017
  - mostly for userspace, loses timing accuracy
- dynamic, “point of interest”
  - probes
- hybrids
  - predefined tracepoints, post-defined payload
  - e.g. dtrace

Other visibility mechanisms

- PC / PMU sampling
  - e.g. Arm Streamline, classic Linux perf
  - limited visibility of program flow
  - sampling IRQ-disabled code is difficult
- Architectural virtualization
  - gem5, Arm Fast Models (FVP), QEMU
  - limited timing accuracy
- Hardware emulation
  - FPGA, RTL simulation, h/w emulator
  - excellent accuracy (up to DDR & I/O) and visibility
  - expensive and/or slow, difficult to interact

FreeBSD

System visibility - compared

	$ bn. cyc	Availability	Speed	Accuracy	Obs'ability
emulation	$200	early	0.1MHz	high	high
simulation	$20	early	1KHz	high	high
"fast RTL"	$2	?	0.01MHz	high	medium
FPGA	10c	medium	10MHz	high	medium
gem5	2c	early(ish)	1MHz	medium	medium
fast models	0.02c	early	100MHz	low/medium	medium
silicon + dtrace	0.001c	late	1GHz	medium	medium

what can we do about this?

Processor trace – hardware assisted

- Two flavors:
  - Arm CoreSight ETM
  - Intel PT
- Highly compressed
- Sequence of branch indicator: taken/not-taken
  - timing optional
- Full address output for indirect branches and exceptions
- To reconstruct control flow, you need the code image
  - e.g. from ELF files, or copy of JIT code cache
  - memory map information for dynamically loaded modules

Processor trace – what's it good for?

- Deep performance analysis
  - deep, cycle accurate analysis of where time is spent in specific routines
  - non-invasive, works for interrupt-disabled code
- Profiling and code coverage
  - sample sequences of branches, similar to Intel LBR
  - can be used with AutoFDO: https://github.com/google/autofdo
- Malware analysis
  - https://www.vmray.com/blog/back-to-the-past-using-intels-processor-trace-for-enhanced-analysis/
- Fuzzing
  - https://github.com/google/honggfuzz
- Post-mortem analysis
  - set up trace in circular "flight recorder" buffer
  - on crash, add trace buffer to crash dump file

System call – two cores

```
163 ] (cpu2) --> mark_held_locks
166 ] (cpu2) @c0055f70:TMB e92d4ff0 PUSH    {r4-r11,lr}
175 ] (cpu2) @c0055f74:TMB 00004682 MOV     r10,r0
176 ] (cpu2) @c0055f76:TMB f8d034b0 LDR     r3,[r0,#0x4b0]
176 ] (cpu2) @c0055f7a:TMB 0000b083 SUB     sp,sp,#0xc
178 ] (cpu2) @c0055f7c:TMB 00002b00 CMP     r3,#0
178 ] (cpu2) @c0055f7e:TMB 0000dd4c BLE     {pc}+0x9c ; 0xc005601a
178 ] (cpu2) @c0055f7c:BR  c005601a
178 ] (cpu2) --> mark_held_locks+0xaa
179 ] (cpu2) @c005601a:TMB 00002001 MOVS    r0,#1
179 ] (cpu2) @c005601c:TMB 0000b003 ADD     sp,sp,#0xc
182 ] (cpu2) @c005601e:TMB e8bd8ff0 POP     {r4-r11,pc}
186 ] (cpu2) @c0056022:BR  c0056096
186 ] (cpu2) --> trace_hardirqs_on_caller+0x62
187 ] (cpu2) @c0056096:TMB 0000b158 CBZ     r0,{pc}+0x1a ; 0xc00560b0
187 ] (cpu2) BR not taken
188 ] (cpu2) @c0056098:TMB f8d5349c LDR     r3,[r5,#0x49c]
190 ] (cpu2) @c005609c:TMB 0000b973 CBNZ    r3,{pc}+0x20 ; 0xc00560bc
190 ] (cpu2) @c005609a:BR  c00560bc
190 ] (cpu2) --> trace_hardirqs_on_caller+0x88
198 ] (cpu2) @c00560bc:TMB 00004639 MOV     r1,r7
198 ] (cpu2) @c00560be:TMB 00004628 MOV     r0,r5
199 ] (cpu2) @c00560c0:TMB f7ffff56 BL      {pc}-0x150 ; 0xc0055f70
199 ] (cpu2) @c00560c0:BR  c0055f70
```

core #1: 33 cycles

```
350 ] (cpu1) --> mark_held_locks
350+] (cpu1) @c0055f70:TMB e92d4ff0 PUSH    {r4-r11,lr}
350+] (cpu1) @c0055f74:TMB 00004682 MOV     r10,r0
350+] (cpu1) @c0055f76:TMB f8d034b0 LDR     r3,[r0,#0x4b0]
350+] (cpu1) @c0055f7a:TMB 0000b083 SUB     sp,sp,#0xc
350+] (cpu1) @c0055f7c:TMB 00002b00 CMP     r3,#0
350+] (cpu1) @c0055f7e:TMB 0000dd4c BLE     {pc}+0x9c ; 0xc005601a
351 ] (cpu1) @c0055f7c:BR  c005601a
351 ] (cpu1) --> mark_held_locks+0xaa
351+] (cpu1) @c005601a:TMB 00002001 MOVS    r0,#1
351+] (cpu1) @c005601c:TMB 0000b003 ADD     sp,sp,#0xc
351+] (cpu1) @c005601e:TMB e8bd8ff0 POP     {r4-r11,pc}
353 ] (cpu1) @c0056022:BR  c0056096
353 ] (cpu1) --> trace_hardirqs_on_caller+0x62
353+] (cpu1) @c0056096:TMB 0000b158 CBZ     r0,{pc}+0x1a ; 0xc00560b0
354 ] (cpu1) BR not taken
354+] (cpu1) @c0056098:TMB f8d5349c LDR     r3,[r5,#0x49c]
354+] (cpu1) @c005609c:TMB 0000b973 CBNZ    r3,{pc}+0x20 ; 0xc00560bc
355 ] (cpu1) @c005609a:BR  c00560bc
355 ] (cpu1) --> trace_hardirqs_on_caller+0x88
355+] (cpu1) @c00560bc:TMB 00004639 MOV     r1,r7
355+] (cpu1) @c00560be:TMB 00004628 MOV     r0,r5
355+] (cpu1) @c00560c0:TMB f7ffff56 BL      {pc}-0x150 ; 0xc0055f70
356 ] (cpu1) @c00560c0:BR  c0055f70
```

core #2: 6 cycles (waypoints only)

System call – two cores

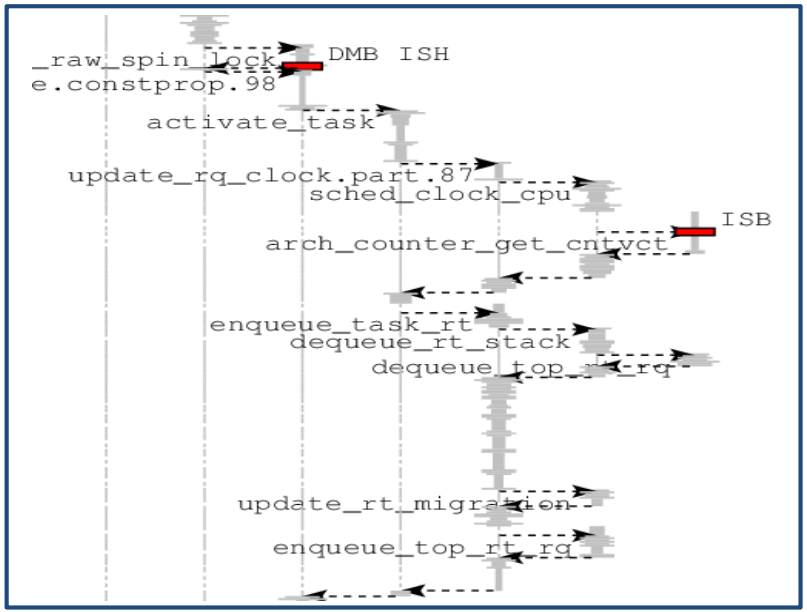
```
31 ] (cpu2) --> vector_swi
32 ] (cpu2) @c000cce0:TMB 0000b092 SUB     sp,sp,#0x48
35 ] (cpu2) @c000cce2:TMB e88d1fff STM     sp,{r0-r12}
44 ] (cpu2) @c000cce6:TMB 000046e8 MOV     r8,sp
45 ] (cpu2) @c000cce8:TMB f3ef8a00 MRS     r10,APSR ; formerly CPSR
47 ] (cpu2) @c000ccec:TMB f08a0a0c EOR     r10,r10,#0xc
50 ] (cpu2) @c000ccf0:TMB f38a8100 MSR     CPSR_c,r10
55 ] (cpu2) @c000ccf4:TMB f8c8d034 STR     sp,[r8,#0x34]
56 ] (cpu2) @c000ccf8:TMB f8c8e038 STR     lr,[r8,#0x38]
56 ] (cpu2) @c000ccfc:TMB f08a0a0c EOR     r10,r10,#0xc
59 ] (cpu2) @c000cd00:TMB f38a8100 MSR     CPSR_c,r10
64 ] (cpu2) @c000cd04:TMB f3ff8800 MRS     r8,SPSR
65 ] (cpu2) @c000cd08:TMB f8cde03c STR     lr,[sp,#0x3c]
66 ] (cpu2) @c000cd0c:TMB f8cd8040 STR     r8,[sp,#0x40]
67 ] (cpu2) @c000cd10:TMB 00009011 STR     r0,[sp,#0x44]
71 ] (cpu2) @c000cd12:TMB f8dfc0ac LDR     r12,{pc}+0xae ; 0xc000cdc0
76 ] (cpu2) @c000cd16:TMB f8dcc000 LDR     r12,[r12,#0]
77 ] (cpu2) @c000cd1a:TMB ee01cf10 MCR     p15,#0x0,r12,c1,c0,#0
83 ] (cpu2) @c000cd1e:TMB e92d500f PUSH    {r0-r3,r12,lr}
86 ] (cpu2) @c000cd22:TMB f049fa09 BL      {pc}+0x49416 ; 0xc0056138
86 ] (cpu2) @c000cd22:BR  c0056138
86 ] (cpu2) --> trace_hardirqs_on
```

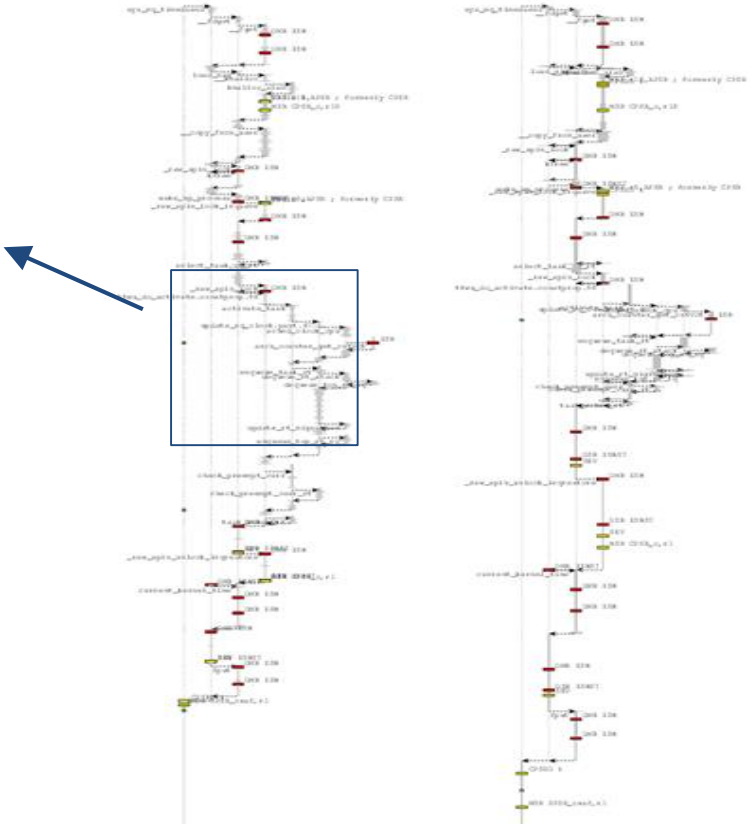
core #1: 55 cycles

```
96 ] (cpu1) --> vector_swi
96+] (cpu1) @c000cce0:TMB 0000b092 SUB     sp,sp,#0x48
96+] (cpu1) @c000cce2:TMB e88d1fff STM     sp,{r0-r12}
96+] (cpu1) @c000cce6:TMB 000046e8 MOV     r8,sp
96+] (cpu1) @c000cce8:TMB f3ef8a00 MRS     r10,APSR ; formerly CPSR
96+] (cpu1) @c000ccec:TMB f08a0a0c EOR     r10,r10,#0xc
96+] (cpu1) @c000ccf0:TMB f38a8100 MSR     CPSR_c,r10
96+] (cpu1) @c000ccf4:TMB f8c8d034 STR     sp,[r8,#0x34]
96+] (cpu1) @c000ccf8:TMB f8c8e038 STR     lr,[r8,#0x38]
96+] (cpu1) @c000ccfc:TMB f08a0a0c EOR     r10,r10,#0xc
96+] (cpu1) @c000cd00:TMB f38a8100 MSR     CPSR_c,r10
96+] (cpu1) @c000cd04:TMB f3ff8800 MRS     r8,SPSR
96+] (cpu1) @c000cd08:TMB f8cde03c STR     lr,[sp,#0x3c]
96+] (cpu1) @c000cd0c:TMB f8cd8040 STR     r8,[sp,#0x40]
96+] (cpu1) @c000cd10:TMB 00009011 STR     r0,[sp,#0x44]
96+] (cpu1) @c000cd12:TMB f8dfc0ac LDR     r12,{pc}+0xae ; 0xc000cdc0
96+] (cpu1) @c000cd16:TMB f8dcc000 LDR     r12,[r12,#0]
96+] (cpu1) @c000cd1a:TMB ee01cf10 MCR     p15,#0x0,r12,c1,c0,#0
96+] (cpu1) @c000cd1e:TMB e92d500f PUSH    {r0-r3,r12,lr}
96+] (cpu1) @c000cd22:TMB f049fa09 BL      {pc}+0x49416 ; 0xc0056138
299 ] (cpu1) @c000cd22:BR  c0056138
299 ] (cpu1) --> trace_hardirqs_on
```

core #2: 203 cycles!

Kernel critical path – two cores





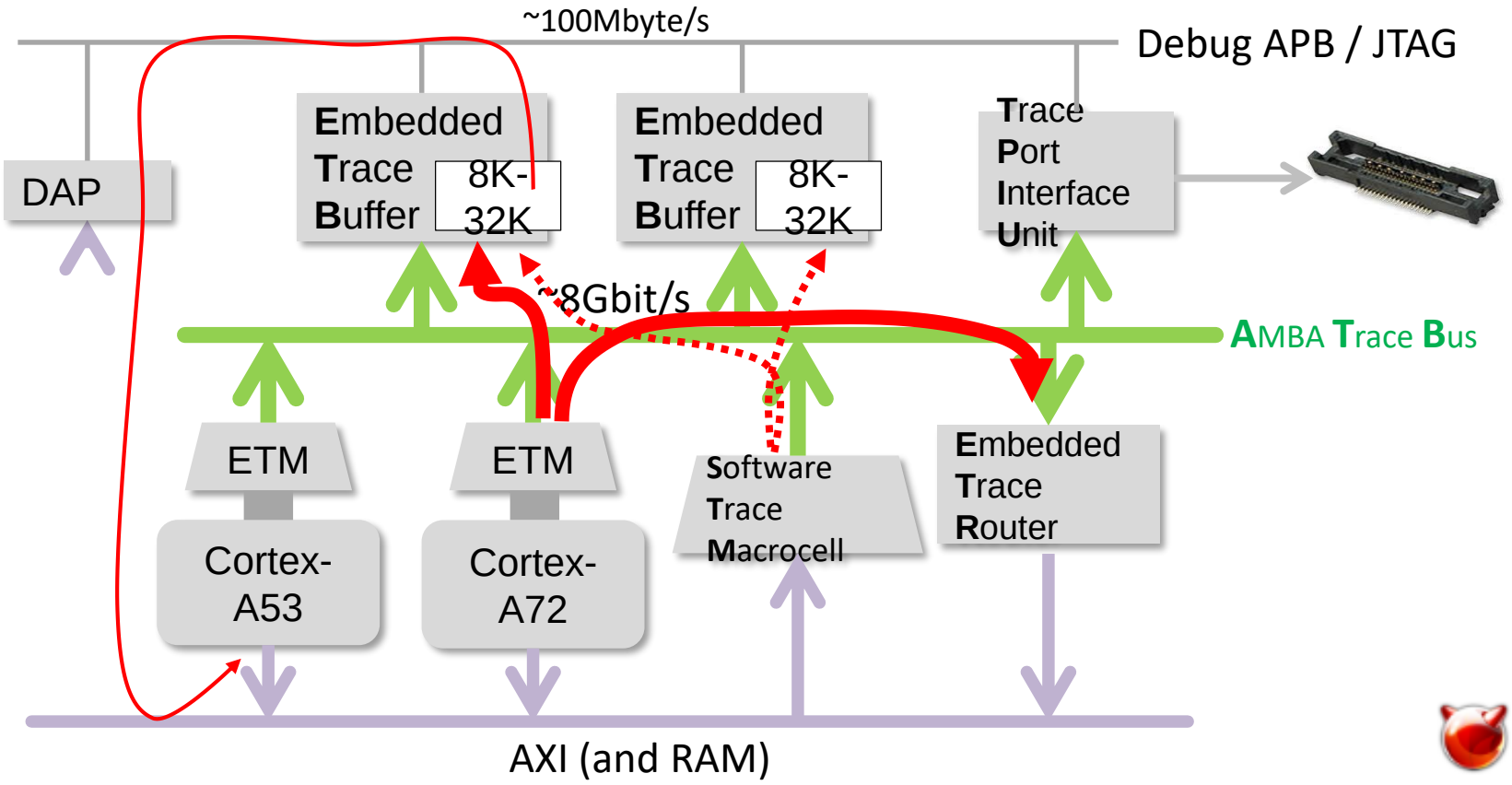
Processor trace - how do I use it?

- In Linux this is now all handled by perf…
- Configure the trace sources
  - via kernel drivers
  - set them up to trace to memory
- Run the workload
  - capturing sideband information as necessary, to reconstruct PC values
- Retrieve the trace
- Decode the trace using decode library
  - need to know configuration
  - need program images or location of code in memory
- Feed the trace into something useful – visualization, analysis, AutoFDO…

Processor trace - decoding

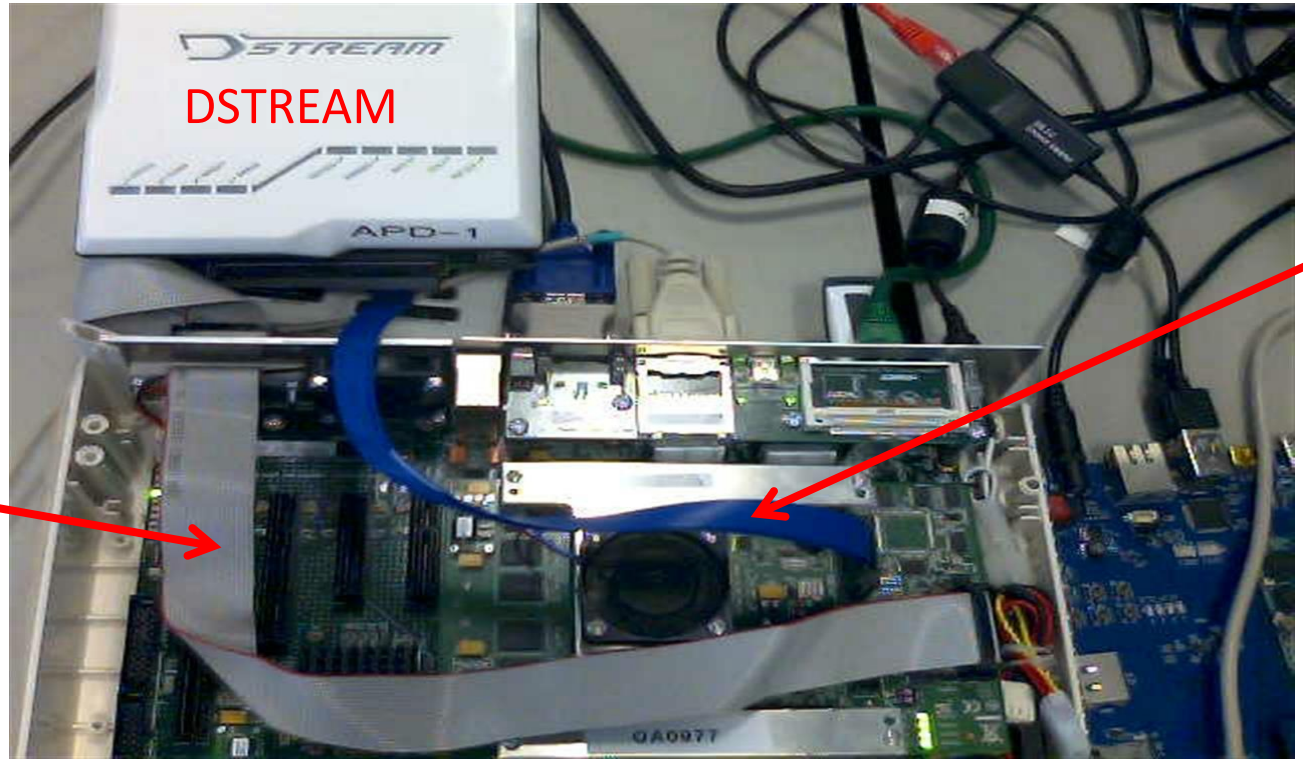
- You will need
  - the trace stream
  - details of how the trace sources were configured
  - images for kernel and program(s)
  - Arm ETM decoder: https://github.com/Linaro/OpenCSD (BSD license)
  - Intel PT decoder: https://github.com/01org/processor-trace (BSD license)
- For dynamically changing address spaces you will also need sideband info
  - timestamped events detailing when the address space mapping changes
  - e.g. loading dynamic libraries
  - context switch
- Trace decode integrates with Linux ‘perf’ tools
  - ETM and PT are well aligned

Arm CoreSight – trace basics



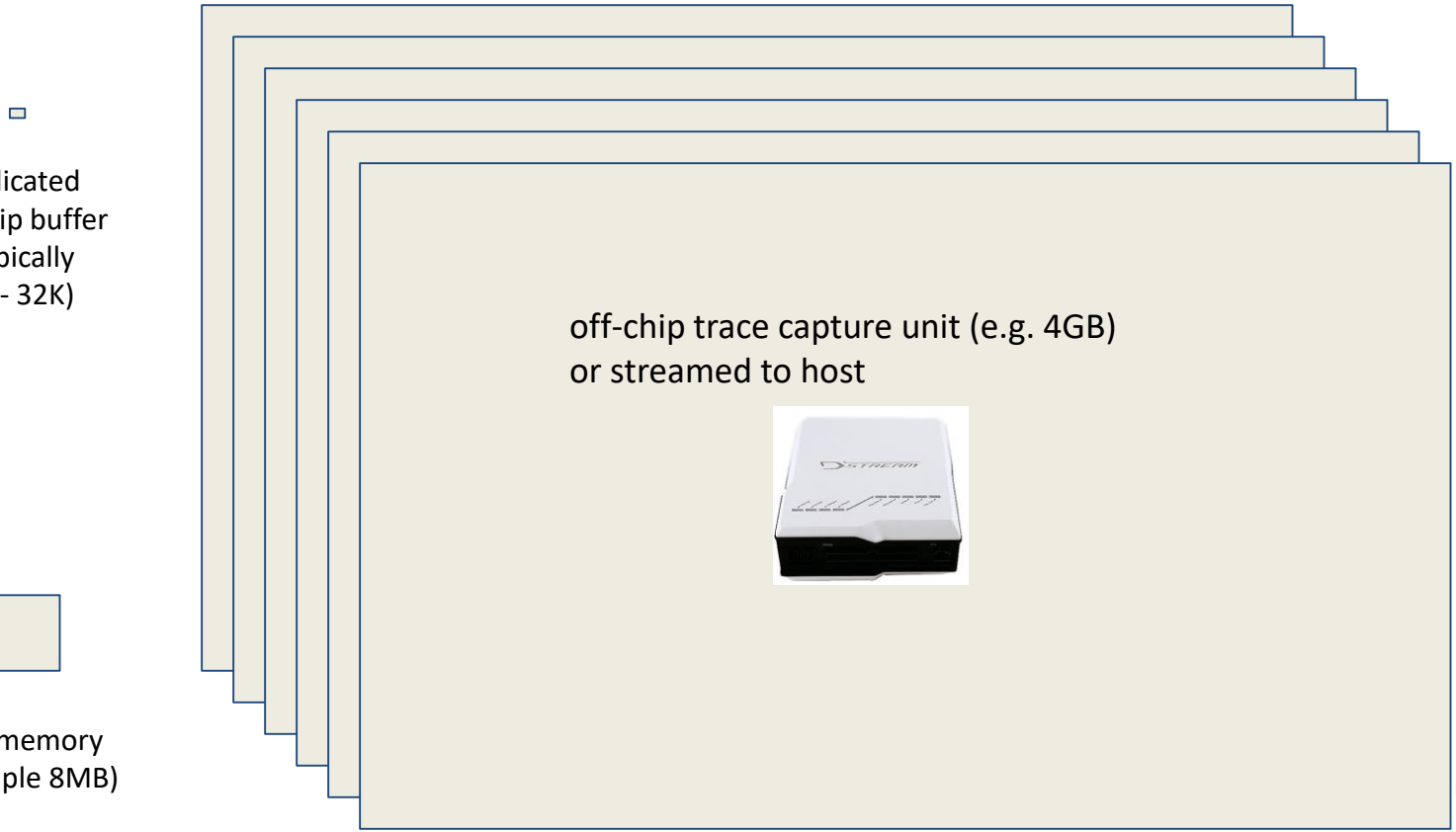
FreeBSD

Arm CoreSight – external capture



FreeBSD

Where is trace captured?



Arm CoreSight trace sources

- Program trace (CoreSight ETM/PTM)
  - branches (“waypoints”) + cycle counts
  - exceptions
  - data trace / instruction trace on embedded CPUs only
- Software trace (CoreSight STM/ITM)
  - messages under program control – conforms to MIPI STPv2
  - messages generated by writing to stimulus area in memory
- Hardware event trace (also CoreSight STM)
  - selected signals, as chosen by SoC designer
  - typically includes CPU interrupt lines
- Common global timestamp for all trace sources

Arm CoreSight device programming

- CoreSight Access Library (CSAL)
  - https://github.com/ARM-software/CSAL : Apache license
- Linux kernel drivers
  - basic framework since Linux 4.1, supports ARMv8
- OS needs to know CoreSight topology for SoC
  - discoverable by probing (but not at boot time)
  - described in vendor documentation (sometimes)
  - Linux Device Tree
  - (ACPI: in progress)
  - SoC-specific recipes may be needed (clocking, power etc.)

So what's new? (if you were here 2 years ago…)

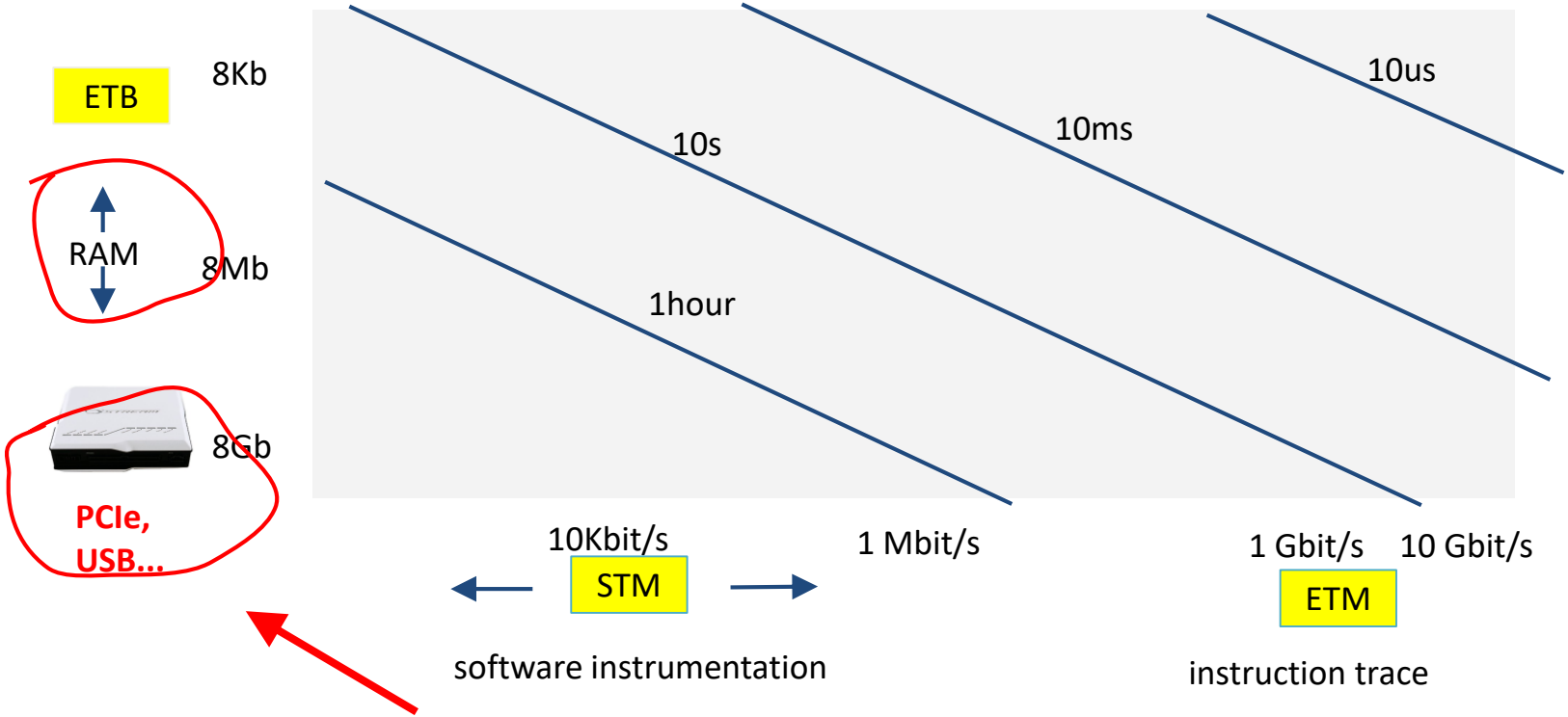
- CoreSight Access Library open-sourced
  - https://github.com/ARM-software/CSAL
- Linux support for CoreSight is more mature
  - integration with perf tools
  - updates to https://github.com/Linaro/OpenCSD
  - documented flow with AutoFDO
- (and DynamoRIO now supports ARMv8…)

For more information…

- ARM CoreSight:
  - http://www.arm.com/products/system-ip/debug-trace/ (product info, aimed at SoC designers)
  - http://infocenter.arm.com/help/index.jsp (technical documentation)
  - http://ds.arm.com/ (debugging tools)
- Linaro blog:
  - https://www.linaro.org/blog/core-dump/coresight-initial-steps-supporting-hw-assisted-tracing-linux-arm-socs/
- Generic STM driver:
  - https://lwn.net/Articles/650245/
- Intel Processor Trace:
  - https://software.intel.com/en-us/blogs/2013/09/18/processor-tracing
- Thank you!

BACKUP

Trace bit rate and buffer size



STM – Software Trace Macrocell

- Injects software-generated messages into the trace stream
- Messages generated by writing to stimulus port area
- Stimulus port area likely to be relatively fast
  - faster than CoreSight device programming registers
- Messages can be timestamped with CoreSight timestamp
- Messages can be blocking or non-blocking
- Message id and options determined by port address
- Stimulus port area can be mapped page by page
  - pages can be mapped into userspace
- Overall cost: generating and writing message data
  - tens of cycles
- No d-cache pollution or inter-core effects!
- Use cases: printf debugging, alternative to ring buffer
- CoreSight STM and Intel's Software Trace Hub are very similar
- Linux now has a generic STM framework covering both