Hardware Traces - The Ultimate Linux Performance Tuning Tool

Dusseldorf, Germany, October 2014

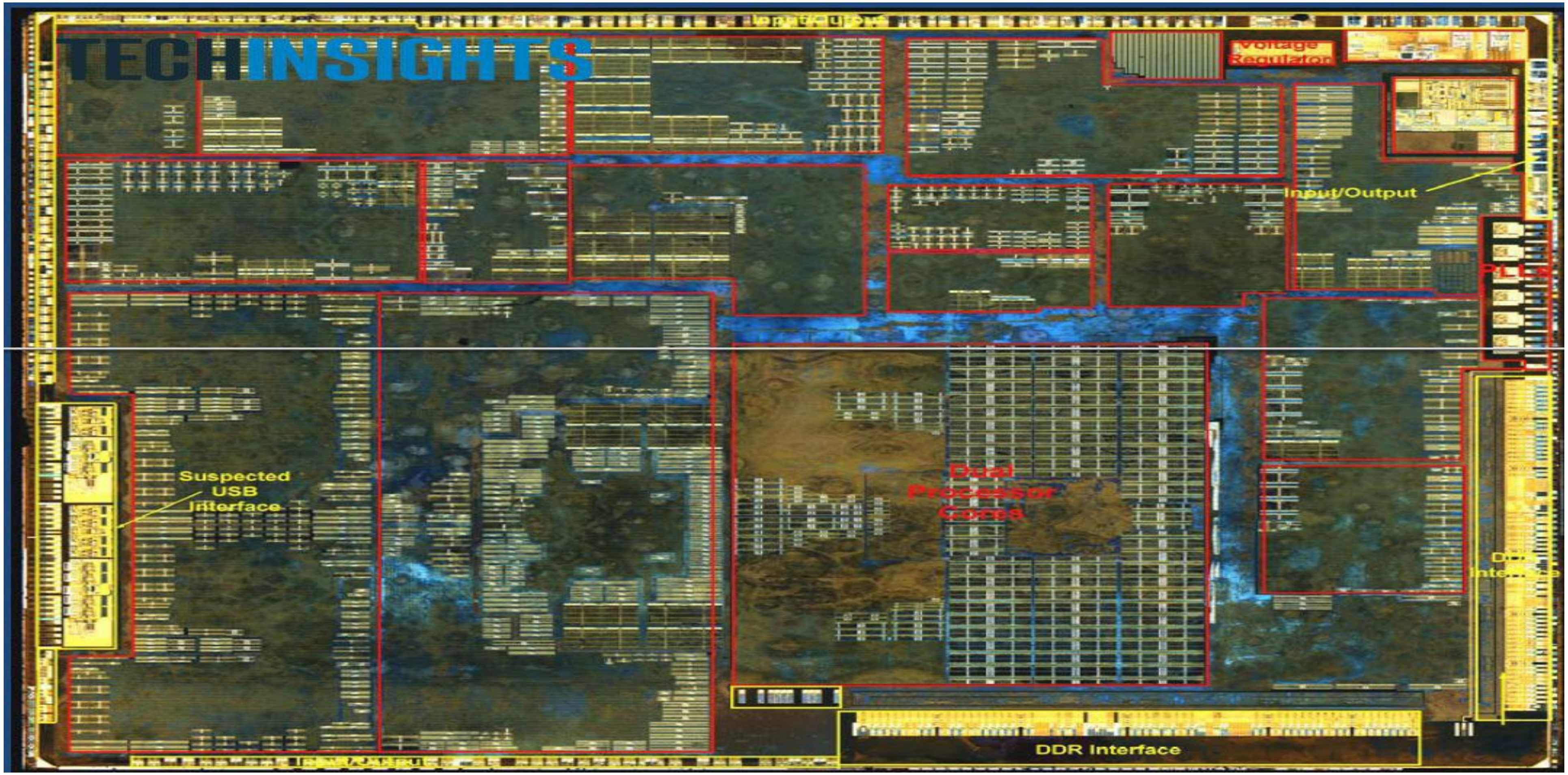
Al Grant (ARM) and Mathieu Poirier (Linaro)

This is a Two Part Presentation

- First half:
  - Brief overview of the Coresight technology
  - The sort of problems it can solve
  - Practical challenges
  - External trace capture

- The second half:
  - Coresight support in the Linux kernel
  - Where we are at in the upstreaming process
  - What we are expected to work on next

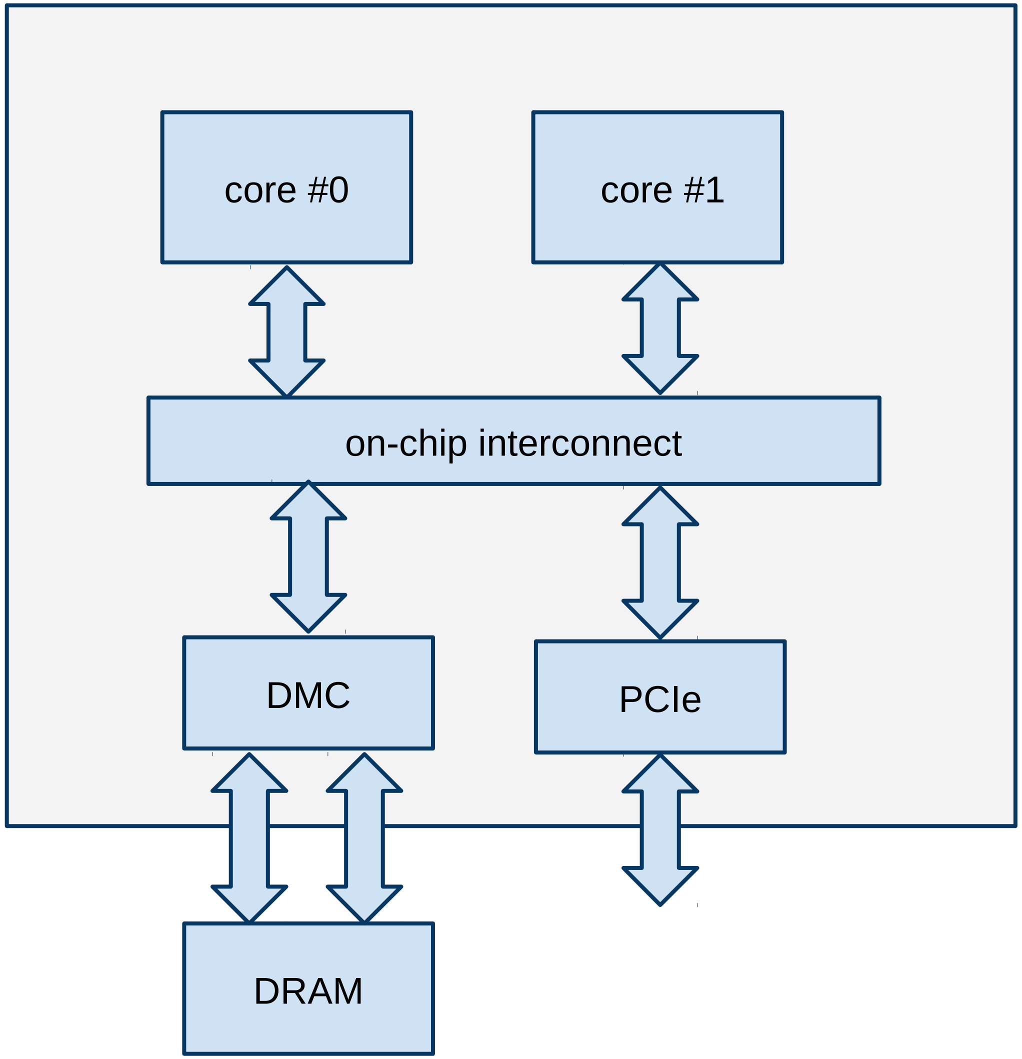
System on Chip (SoC)



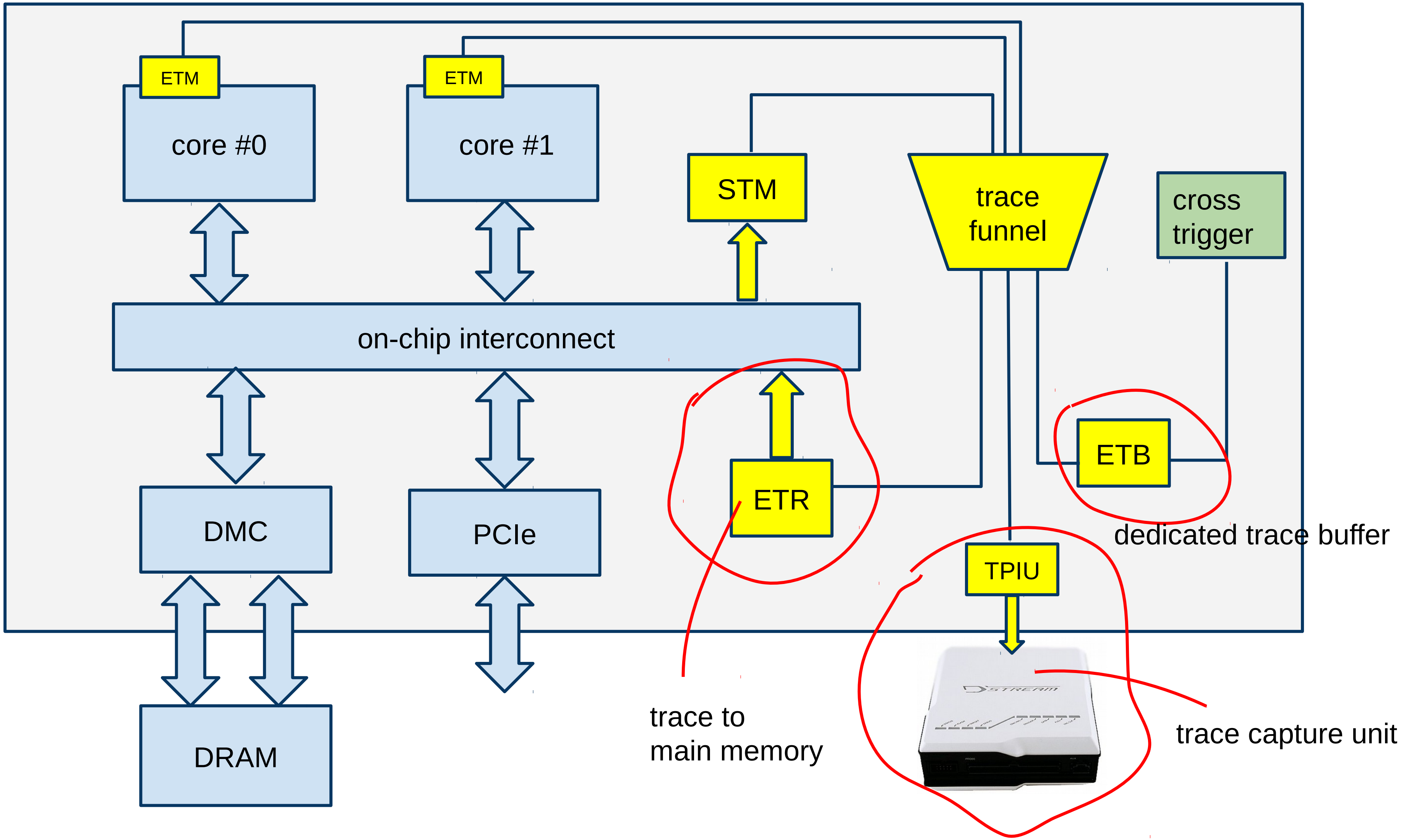
(image courtesy UBM TechInsights via AnandTech)

How do we debug this?
How do we observe what’s going on?
How do we relate hardware events to software?

SoC simplified



CoreSight on-chip debug and trace



dedicated trace buffer

trace to main memory

trace capture unit

CoreSight components can be accessed via
- external JTAG (not shown)
- on-chip memory-mapped access

What can be traced?

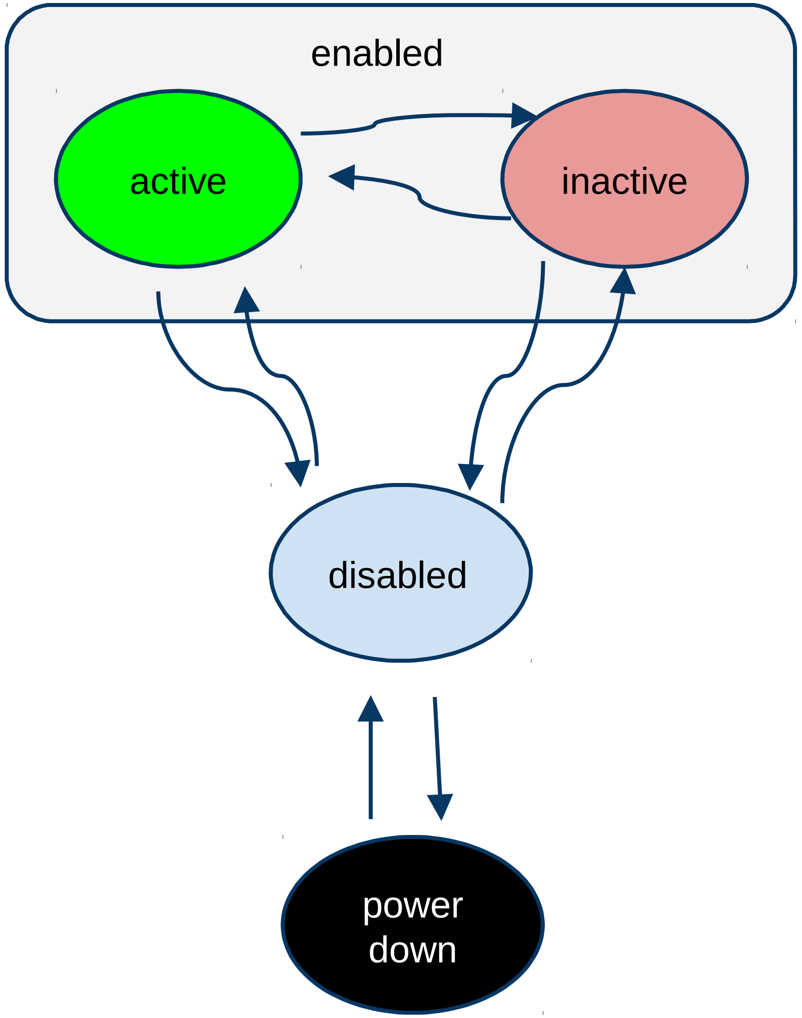
- Core instruction trace (ETM/PTM)
  - explained in more detail later
  - about ~1Gbit/s to ~10Gbit/s per core when active
- Software instrumentation
  - by writing to CoreSight STM or ITM
  - STM uses MIPI STPv2 trace protocol
- Selected hardware events
  - if input to CoreSight STM (SoC specific)
- other SoC-specific trace sources
  - might or might not be exposed for end-user use
- each trace source is identified by a 7-bit identifier
- multiplexed together by trace funnels

ETM/PTM naming (a digression)

- Each ETM/PTM unit traces the activity of a single core
  - all ARM cores have had an ETM, since forever...
- ETM, PTM, what’s the difference?
  - ETM v3.x (example: ARM1176, Cortex-A7)
    - instruction trace: one trace element per instruction
  - PTM v1.x (example: Cortex-A9, Cortex-A15)
    - program flow trace: one trace element per branch
    - allows trace to keep up with GHz core speeds
  - ETM v4 (example: Cortex-A53, Cortex-A57)
    - offers both options, but for A-profile is more PTM-like
- Think of PTM as “the one in between ETMv3 and ETMv4”
  - we can just talk about ETM from now on...

Programming the ETM

- ETM outputs trace when **active**
- Active/inactive state is determined by how ETM is programmed:
  - address comparators
  - user/privileged state
  - start/stop events
  - sequencer state
  - CONTEXTID/VMID matching
- ETM can be programmed only when disabled
- Each core has its own ETM



Decoding ETM

- ETM is highly compressed
  - Logically: address of every executed instruction
  - Actually: 1 bit per conditional branch
- To reconstruct instruction stream, we need the code
- For kernel, mostly easy
  - code modification (dynamic ftrace)
  - loadable kernel modules
- For userspace, need to know current address space and map of that address space
  - CONTEXTID can help but use is optional
- Generally, we need **metadata**

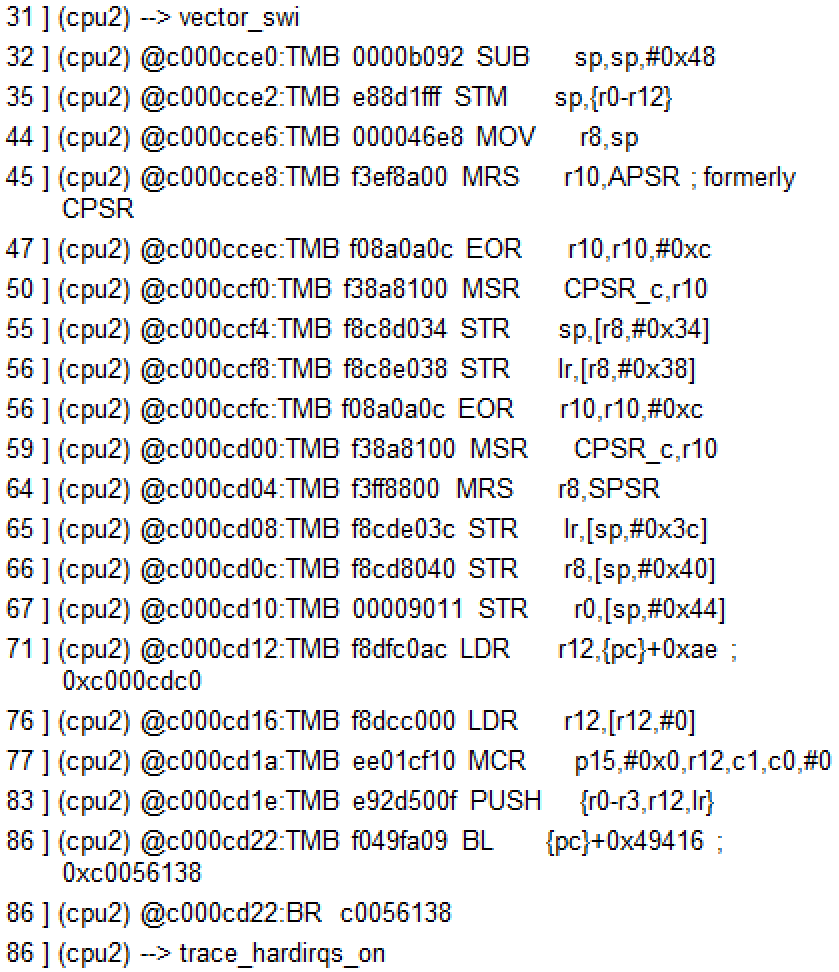
So what can we do with ETM?

- Targeted tracing for performance investigations
  - use ETM filtering to activate trace round region of interest
- Sampling profiler/coverage tool
  - repeatedly capture trace fragments
  - accurately measures basic block execution times
  - use “shotgun sequencing” to construct a larger profile
- First-failure data capture
  - capture rolling trace into buffer from boot time onwards
  - stop capture when fault is detected

ETM strengths and challenges

- Strengths
  - it’s non-invasive
    - can be enabled all the time
    - can be programmed to activate/deactivate itself round regions of interest
  - traces interrupt-disabled code, exceptions etc.
  - traces multiple cores
- Challenges
  - decoding requires access to program image
  - high bit rate might quickly fill up buffer

ETM for kernel tuning



```
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
```

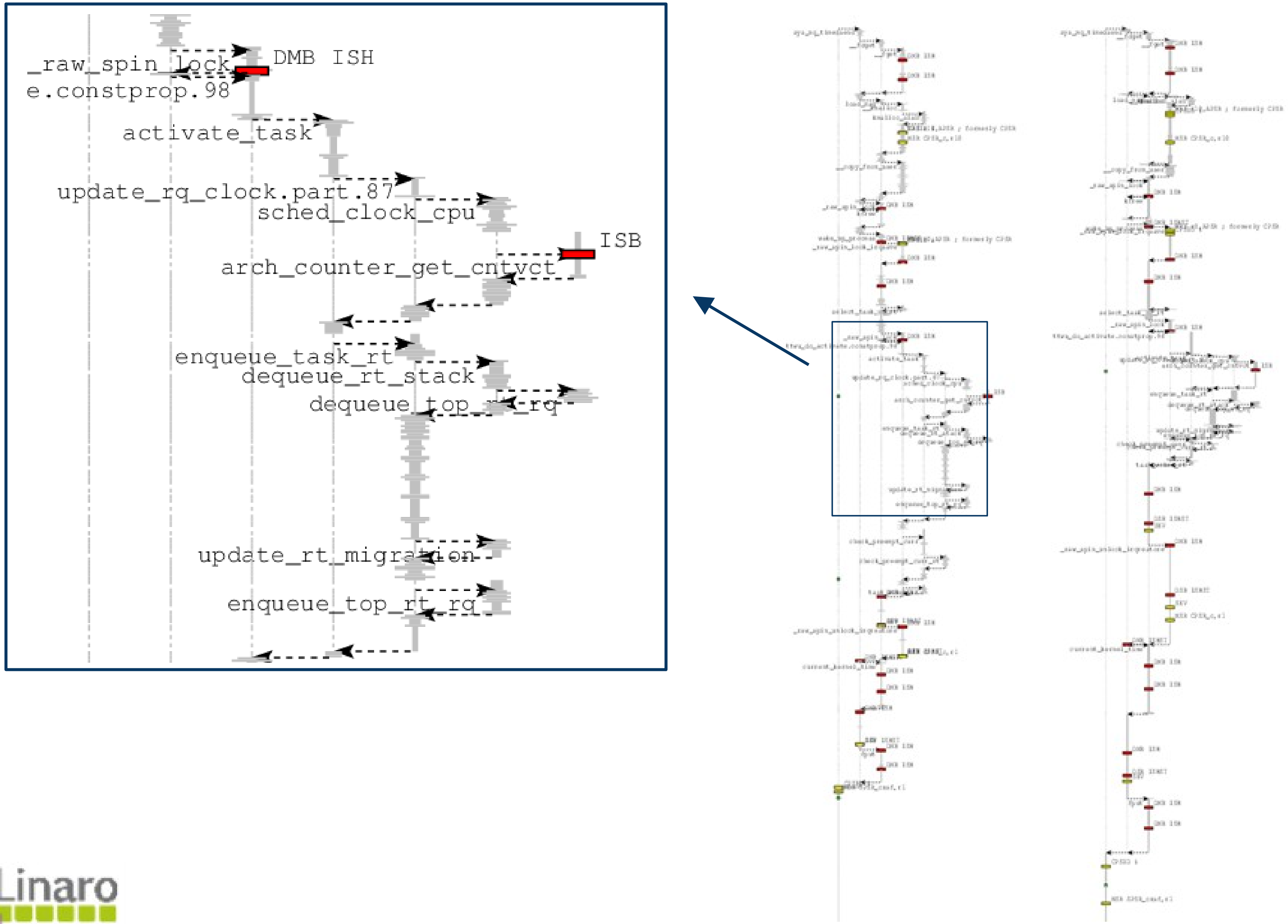
```
96 ] (cpu1) --> vector_swi
96+] (cpu1) @c000cce0:TMB 0000b092 SUB     sp,sp,#0x48
96+] (cpu1) @c000cce2:TMB e88d1fff STM     sp,{r0-r12}
96+] (cpu1) @c000cce6:TMB 000046e8 MOV     r8,sp
96+] (cpu1) @c000cce8:TMB f3ef8a00 MRS     r10,APSR ; formerly
     CPSR
96+] (cpu1) @c000ccec:TMB f08a0a0c EOR     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
```

A7: 55 cycles

A15: 203 cycles!

(PTM: waypoint timings only)

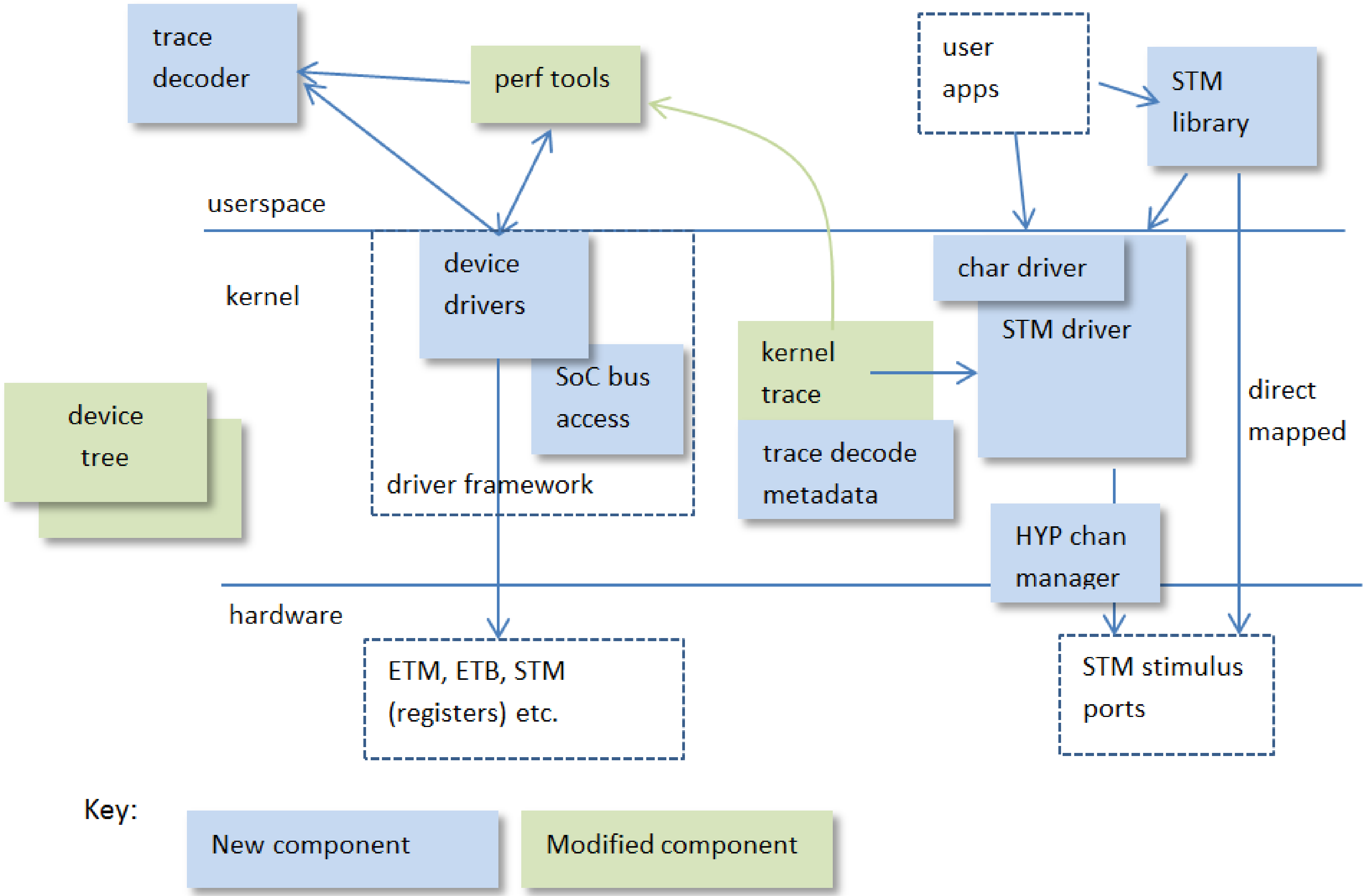
ETM for kernel tuning



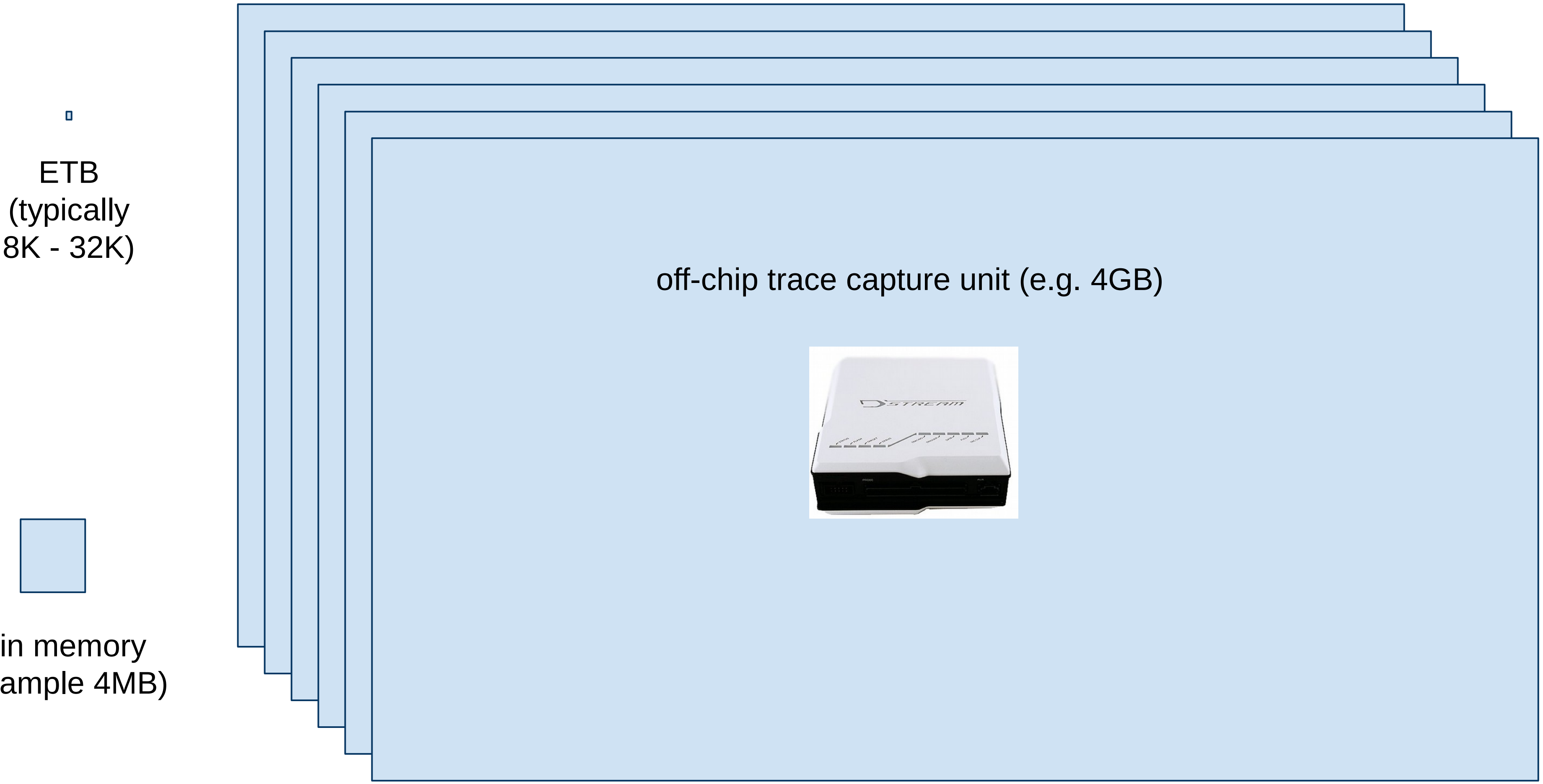
STM - software trace macrocell

- 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 address
- Stimulus port area can be mapped page by page
  - pages can be mapped into userspace
- Overall cost: generating and storing message data
  - tens of cycles
- No d-cache pollution!

STM - software trace macrocell



Where can trace be captured?



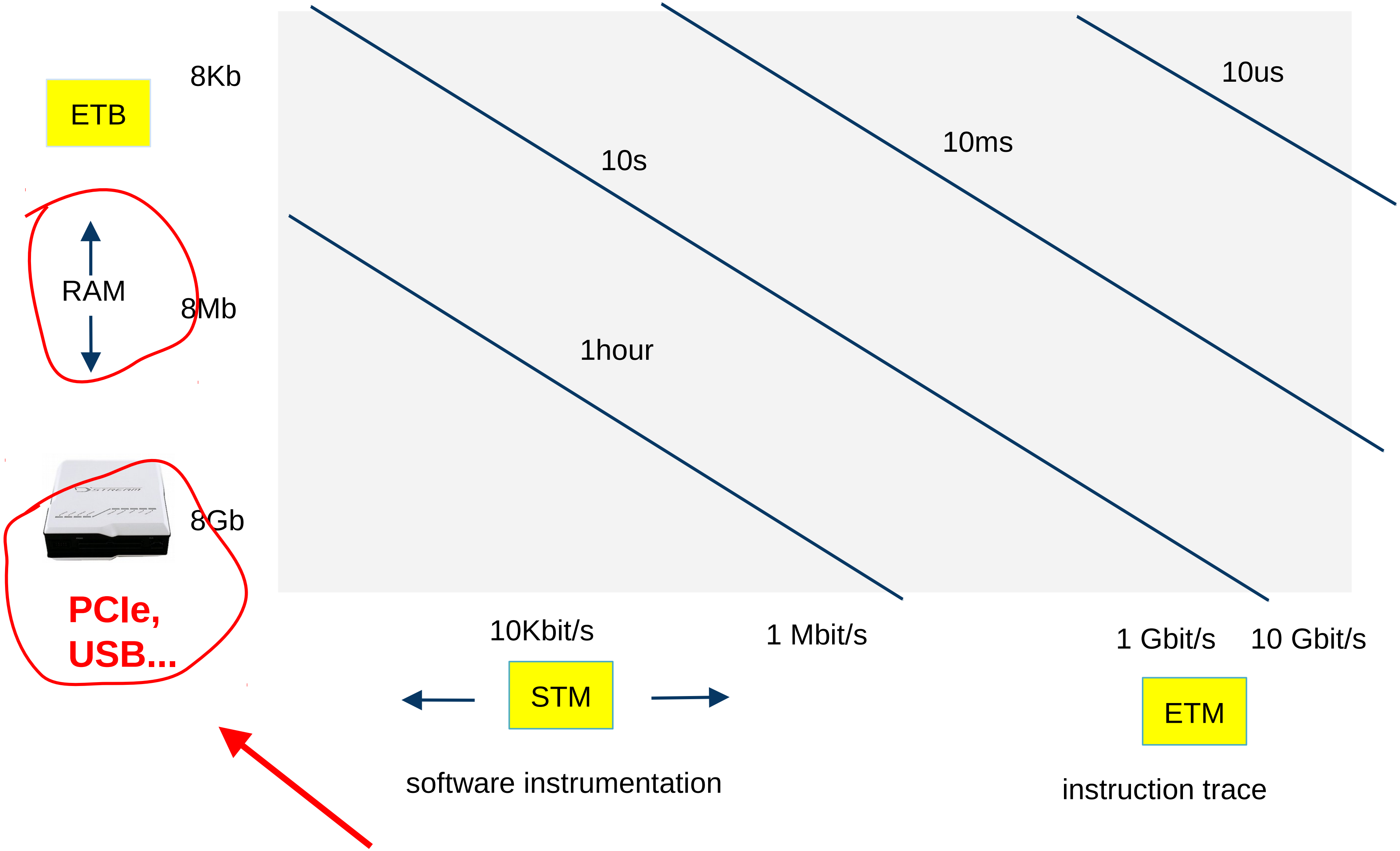
ETB
(typically
8K - 32K)

off-chip trace capture unit (e.g. 4GB)

main memory
(example 4MB)

but… trace can be read out of in-memory buffers and saved on disk etc.

Trace bit rate and buffer size



CoreSight trace is becoming more practical and accessible!

Coresight Support in the Linux Kernel

- Linaro has been working on Coresight since March 2014
- Started from the initial framework submitted by Pratik Patel in December of 2012 [1]
- The framework provides support for:
  - source: ETM v3.3 to v3.5 and PTM v1.0, v1.1
  - link: 8 port funnel and non-configurable replicator
  - sink: ETBv1.0, TPIU and TMC (Trace Memory Ctrl)

- Support for STM and CTI will be submitted when the base framework is accepted

Coresight Framework Highlights

- Provides an easy integration via DT for any platform with generic components
- Plenty of flexibility for addition of new, non-generic devices
- Access to configuration registers via sysfs
- Processor state (hibernation) decoupled from ETM/PTM configuration
- Multiple configuration of source and sink is supported
- Provides interface for reporting the status of various component and the gathering of *metadata*

Where to Get the Code

- Official Coresight Branch on git.linaro.org:
  - https://git.linaro.org/kernel/coresight.git
  - Always look at the master branch for the latest code
  - Code is always based on the latest kernel release
  - Everything is under *drivers/coresight*
  - Menuconfig option is under

    “Kernel Hacking/Coresight Tracing Support”

- Earlier submission and initial RFC are also present
- Google “Coresight framework and drivers” for discussions

Documentation and Examples

- Documentation lives under:
  - Documentation/trace/coresight.txt
  - Documentation/devicetree/bindings/arm/coresight.txt

- Two sessions at Linaro Connect [2,3]

- Example of a simple use case scenario and how to use the framework is documented here [4]
  - That example is for ARM’s TC2 board but has been proven to work on Huawei’s D01 platform.
- There is also a more generic blog post here [5]

What is Next

- Priority is on upstreaming of framework and support for base components

- Concurrently:
    - Support for Qualcomm’s APQ80x4 and TI’s UEVM5432
    - Support for STM32 on ARM’s V8 Juno platform

- On a longer term, drivers for
    - Cross Trigger Interface (CTI)
    - STM500

Questions and Comments

[1].http://lists.infradead.org/pipermail/linux-arm-kernel/2012-December/138028.html
[2].http://lcu14.zerista.com/event/member/137703
[3].http://lcu14.zerista.com/event/member/137708
[4].https://wiki.linaro.org/WorklingGroups/Kernel/Coresight/traceDecodingWithDS5
[5].http://www.linaro.org/blog/core-dump/coresight-initial-steps-supporting-hw-assisted-tracing-linux-arm-socs/