Background

- This is a technical talk for developers
  - not official statements, product roadmap etc.
- I use Arm examples because it's what I know, but there are often equivalents in other architectures
- Our challenges are not specific to any architecture:
  - can we get insights from low-level hardware telemetry through usable, architecture-agnostic tools, in a way that help end users solve real performance problems in the real world?
  - can we work together on common challenges e.g. tooling, security, uncore, methodologies?

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, the path to memory…
- Use of system resources: memory, power

Software trace

- programmer-inserted (printf debugging)
  - problem: no standard language mechanism
- compiler-inserted
  - e.g. `gcc -pg`
- DBI: Pin, DynamoRIO: Dynamic binary instrumentation
  - DynamoRIO: https://github.com/DynamoRIO/dynamorio
- dynamic
  - ftrace, kprobes, libpatch
- hybrids
  - predefined tracepoints, post-defined payload
  - e.g. dtrace, ftrace, eBPF

Software trace - what are its limitations?

- The probe effect
- Instrumentation affects:
  - Timing
  - Cache / TLB contents
  - PMU hardware events
- Difficult to go to the lowest levels
  - interrupt-disabled code
- Heisenbugs may stop happening entirely
- May struggle with JIT

The ideal

- Just trace everything!
- Hardware emulator
  - expensive
- Software simulation
  - very very slow (~1000 cycles/s)
  - it takes weeks to boot Linux
- One core is slow, never mind 128 cores
- Debugging wave files is hard...
- You need the source of the hardware...
- But... if you can isolate a specific performance issue, *someone* will be able to investigate at this level

picture sourced from internet, nothing proprietary here...

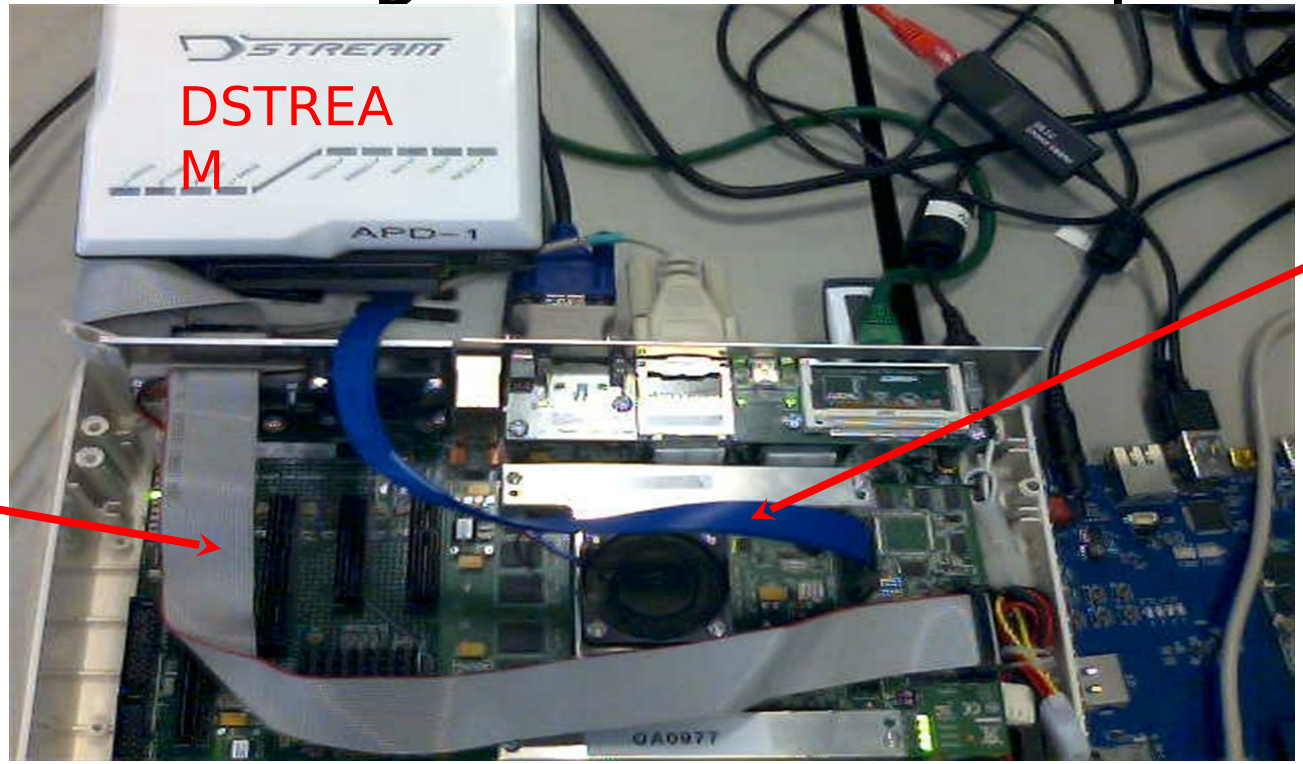
Other visibility mechanisms

- PC / PMU sampling
  - e.g. Arm Streamline, classic Linux perf
  - limited visibility of program flow
  - sampling IRQ-disabled code is difficult
  - variety of hardware PMU events, many model-specific
- Architectural virtualization
  - QEMU, Simics
  - limited timing accuracy
- Architectural with cache/TLB/memory model
  - DynamoRIO with drcachesim, QEMU with cache plugin
- Full system performance model
  - gem5
  - closer to accurate, but may lack architecture completeness
  - research tool, not so good for analyzing production workloads

Processor trace – hardware assisted

- Vendor-specific instruction trace
  - Arm ETM/ETE
  - Intel PT
- Highly compressed
- Direct branches indicated with a branch indicators: 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

Arm CoreSight – external capture

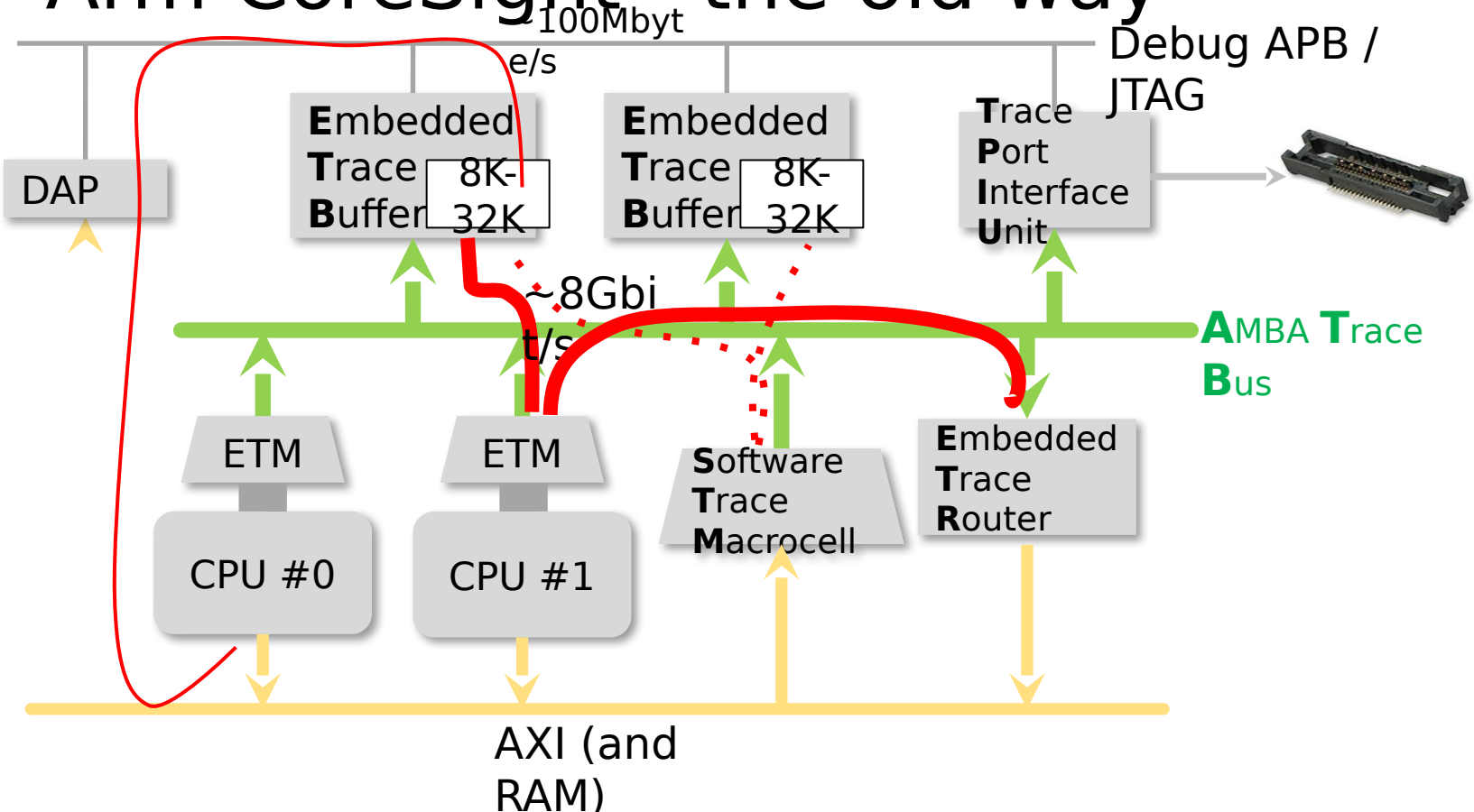


DSTREAM

JTAG

trace

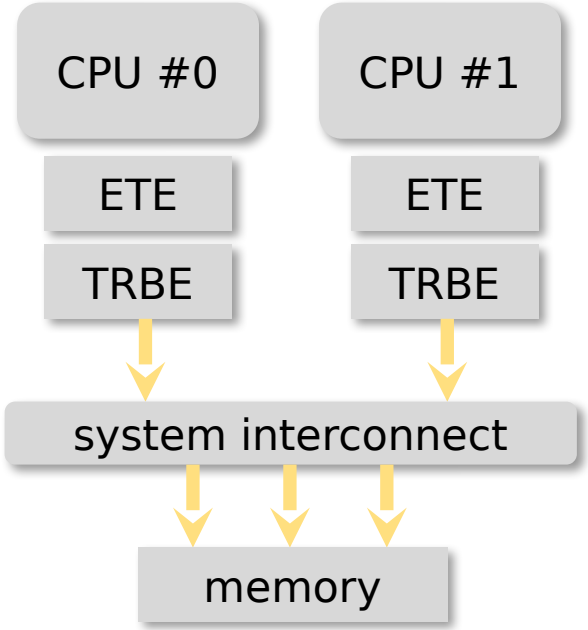
Arm CoreSight - the old way



Arm ETE/TRBE - the new way (ARMv9 onwards)

- Each CPU has an ETE trace unit
  - essentially similar to ETM
- Trace is written by TRBE to virtual memory
- Trace buffer context can be managed as part of process/thread context
- Kernel's job becomes a lot easier

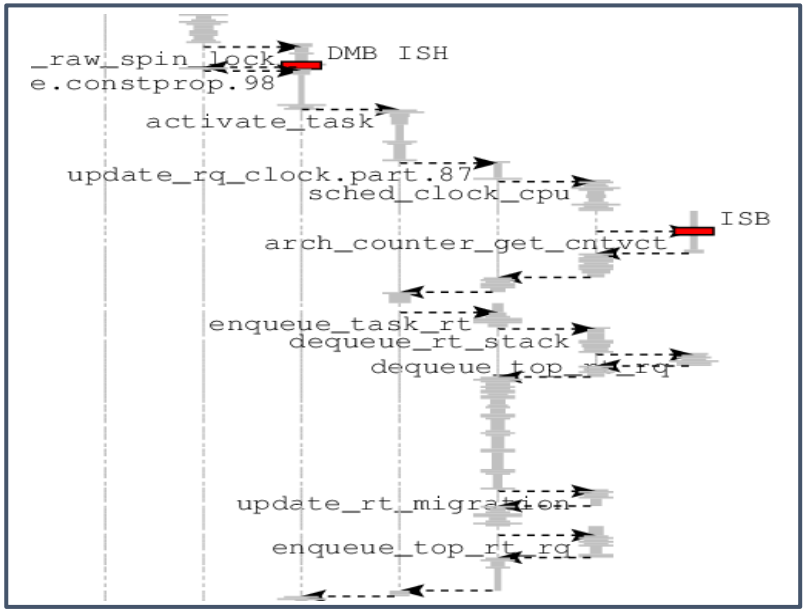
- No change to Linux perf tools
- Trace should become more ubiquitous (but see comments on security later…)

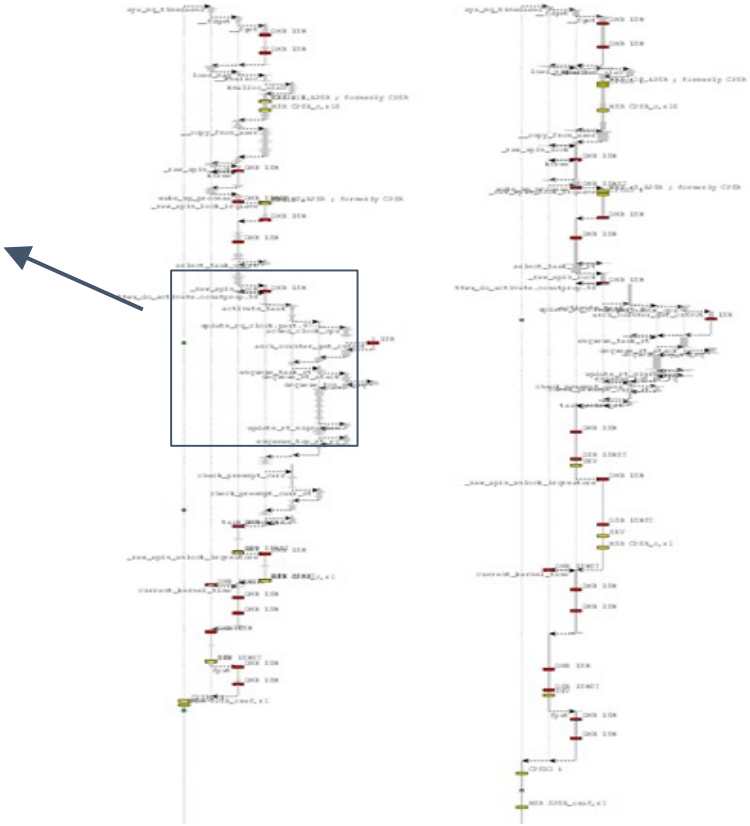


Processor trace – what's it good for?

- 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

Kernel critical path – two cores





Processor trace - how do I use it?

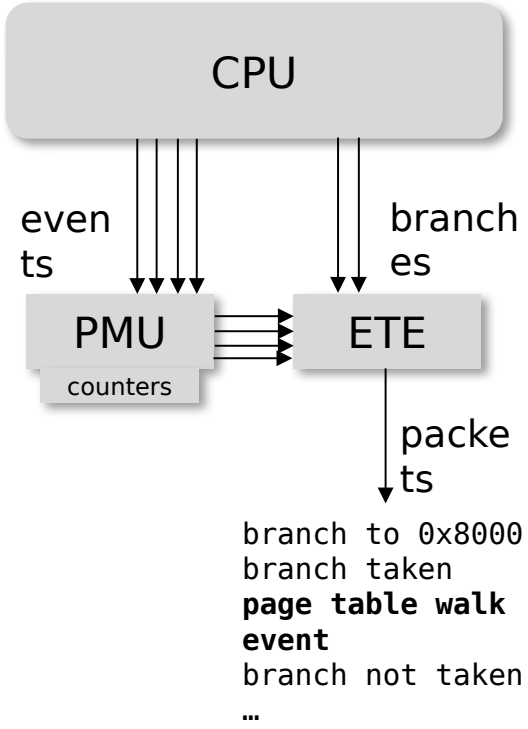
- In Linux this is now all handled by perf...
  - perf record -e cs_etm// ...
- Filters can be used to set address range or start/stop address
- Kernel will configure the trace sources
- Run the workload
  - capturing sideband information as necessary, to reconstruct PC values
- Retrieve the trace
- Decode the trace using decode library e.g. OpenCSD
  - need to know trace configuration - recorded in PERF_RECORD_AUXTRACE_INFO
  - need program images or location of code in memory
  - "perf inject" (build with libopencsd) can decode trace back into samples
- 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/ETE decoder: https://github.com/Linaro/OpenCSD (BSD license)
  - Intel PT decoder
- For dynamically changing address spaces you will also need sideband info
  - PERF_RECORD_MMAP etc. for loading dynamic libraries
  - PERF_RECORD_TEXT_POKE for kernel self-modifying code
  - context switch
- No trace decoding in the kernel! Trace is decoded in userspace
- "perf inject" converts trace into branch records
  - massively increases size of trace file, unless samping used

Tracing hardware performance events

- Instead of periodically sampling the counters, why not trace the actual events as they occur?
- On Arm, PMU exports events to hardware trace
  - events in JSON at https://github.com/ARM-software/data
  - use the "**`trace_lsb`**" number
- Trace up to four bits per cycle
  - multi-bit events: in ETM, appear as multiple bits, in ETE, are OR-ed together
- Events are traced in Event packet, along with branches
- Some skew - events are not precisely located with respect to instruction which caused them
- Finer-grained than counting, less intrusive than sampling
- Not currently supported by Linux perf tools
  - but can be enabled with the new "complex configurations" feature



```
[16] 0  0000ffffbad85450  910003fd  MOV      x29,sp                          |                  __kernel_clock_gettime+0x20
[16] TIME 752 cc=2 - Cycle Count: 2 (Timestamp packet (43))
[16] 0  0000ffffbad85314  14000002 +B         {pc}+8 ; 0xbad8531c
[16] events: ISB
[16] 0  0000ffffbad8531c  b9400004  LDR      w4,[x0,#0]
[16] 0  0000ffffbad85320  3707ffc4 -TBNZ     w4,#0,{pc}-8 ; 0xbad85318
[16] 0  0000ffffbad85324  d50339bf  DMB      ISHLD
[16] 0  0000ffffbad85328  b9400403  LDR      w3,[x0,#4]
[16] TIME 753 cc=2 - Cycle Count: 2 (Timestamp packet (45))
[16] 0  0000ffffbad8532c  34000063 +CBZ      w3,{pc}+0xc ; 0xbad85338
[16] TIME 754 cc=11 - Cycle Count: 11 (Timestamp packet (56))
[16] 0  0000ffffbad85338  d5033fdf +ISB
[16] TIME 755 cc=32 - Cycle Count: 32 (Timestamp packet (88))
[16] events: ISB
[16] 0  0000ffffbad8533c  d53be043  MRS      x3,CNTVCT_EL0
[16] TIME 756 cc=12 - Cycle Count: 12 (Timestamp packet (100))
[16] 0  0000ffffbad85340  d5033fdf +ISB
[16] TIME 757 cc=21 - Cycle Count: 21 (Timestamp packet (121))
[16] 0  0000ffffbad8534c  b7ffff23 -TBNZ     x3,#63,{pc}-0x1c ; 0xbad85330 |                  <[vdso]>+0x34c
```

Statistical Profiling (Arm SPE)

- SPE is an optional feature of the Arm architecture from v8.2 onwards
- SPE traces samples of individual instructions, with detail for performance analysis
- SPE runs in the background: samples do not interrupt the running thread
- Samples are traced to memory, bypassing cache, but are coherent
- SPE sample records can include:
  - timestamp
  - **precise** instruction address
  - instruction latency
  - **data virtual address**
  - **data source** information e.g. whether a load hit in L1, L2, other socket, DRAM…
  - events e.g. L1 miss, branch mispredict, implementation-defined events
- SPE samples can be filtered:
  - sample type e.g. load, store, branch
  - minimum latency e.g. select only accesses >= 10 cycles
  - events e.g. select only mispredicted branches

Arm SPE integration in Linux

Target system

perf record

gatord

perf_event_open()

Linux kernel

perf subsystem

core SPE

core SPE

core SPE

core SPE

System memor

buffer

buffer

buffer

buffer

Using Arm SPE with Linux perf tools

- SPE appears as a new “PMU” (arm_spe) in the Linux perf subsystem
- This PMU can be discovered, and opened via `perf_event_open()`
- Kernel will allocate a buffer in system memory
- Kernel will program SPE filtering options to match user’s request
- Kernel will periodically collect data from the buffer
- SPE data is returned as an AUX buffer

- "perf record" can't use SPE and ETM at the same time. Not a hardware or perf_event_open limitation.
  - You can do "perf record -e arm_spe// -- perf record -e cs_etm// …"
  - Are we testing the limits of perf and perf.data?

Using Arm SPE with Linux perf tools

- SPE can be accessed directly
  - `perf record -e arm_spe// -- ./bench`
- SPE filtering options can be specified
  - `perf record -e arm_spe/load_filter=1,min_latency=10,ts_enable=1/ -- ./bench`
- SPE data is captured as raw data in AUX records
  - can be viewed with “`perf record –D`”
  - data format is simple and easy to process: decode doesn’t need access to program image
  - although PC samples has the usual problem for JITted code
- SPE now supported in "perf mem" and "perf c2c"
  - see ACME's "State of the Linux tracers" trace from Tuesday
  - breakdown of which memory level loads were satisfied from, load latency etc.
  - https://lore.kernel.org/lkml/20220530114036.3225544-1-leo.yan@linaro.org/

Architectural vs. implementation-specific

Architected	Implementation-specific
CPU: functionality of instructions for any given architecture feature.	CPU: which architecture features are implemented - use ID registers to check (or hwcaps in userspace). Instruction timings.
Trace: trace format. Trace filtering capabilities.	Trace: trace of speculative execution In v8: all details of trace programming and collection. PMU event number mappings to trace event packets. In v9: IRQ number for trace buffer overflow
PMU: architected events e.g. INST_RETIRED, L1D_CACHE_REFILL.	PMU: implementation defined events. Exact behavior of some architected events. See https://github.com/ARM-software/data Number of counters (it's usually 6, but not always)
SPE: basic features.	SPE: whether present or not. IRQ number for SPE buffer overflow. Additional impl def events. Meaning of "data source".

Heterogeneous systems (e.g. "big.LITTLE") will enforce consistency in some areas but not others

Hardware trace: some challenges

- Security
  - must not leak kernel data, other thread data, kernel address (break KASLR) etc. - side-channel threats!
- Single use at a time
  - PMU counters can be partitioned and time-multiplexed; other hardware features less so
- Virtualization
  - security/stability/portability concerns: h/w telemetry missing from non-bare-metal instances
  - KVM support for SPE and ETE/TRBE are work-in-progress; S2 translation faults complicate things
- Software and tools
  - how do we capture multiple streams of trace?
  - how can UIs designed for event trace (microseconds), best handle instruction trace (nanoseconds)?
- Variation between architectures and implementations
- Uncore even less standardized