perf_events status update

Stephane Eranian
Google, Inc

CSCADS workshop
Snowbird, UT
August 2010

Agenda

- signal update
- perf_events update
- perf tool update
- libpfm4 update

Signal update

- POSIX: cannot target async signals to a specific thread

- issue for self-sampling multithreaded workloads
  - SIGIO must go to thread where the event occurred

- raised the issue on LKML
  - Zjilstra proposed a patch to extend fcntl()

- 2.6.32: new F_SETOWN_EX/F_GETOWN_EX commands
  - may not yet be in the system header files

Signal example

```
#ifndef F_SETOWN_EX
#define F_SETOWN_EX     15
#define F_GETOWN_EX     16

#define F_OWNER_TID     0
#define F_OWNER_PID     1
#define F_OWNER_PGRP    2
#endif

struct f_owner_ex {
        int     type;
        pid_t   pid;
};

struct f_owner_ex fown;

fown.type = F_OWNER_TID;
fown.pid  = gettid();

fcntl(fd, F_SETOWN_EX, &fown);
```

Perf_event key design choices

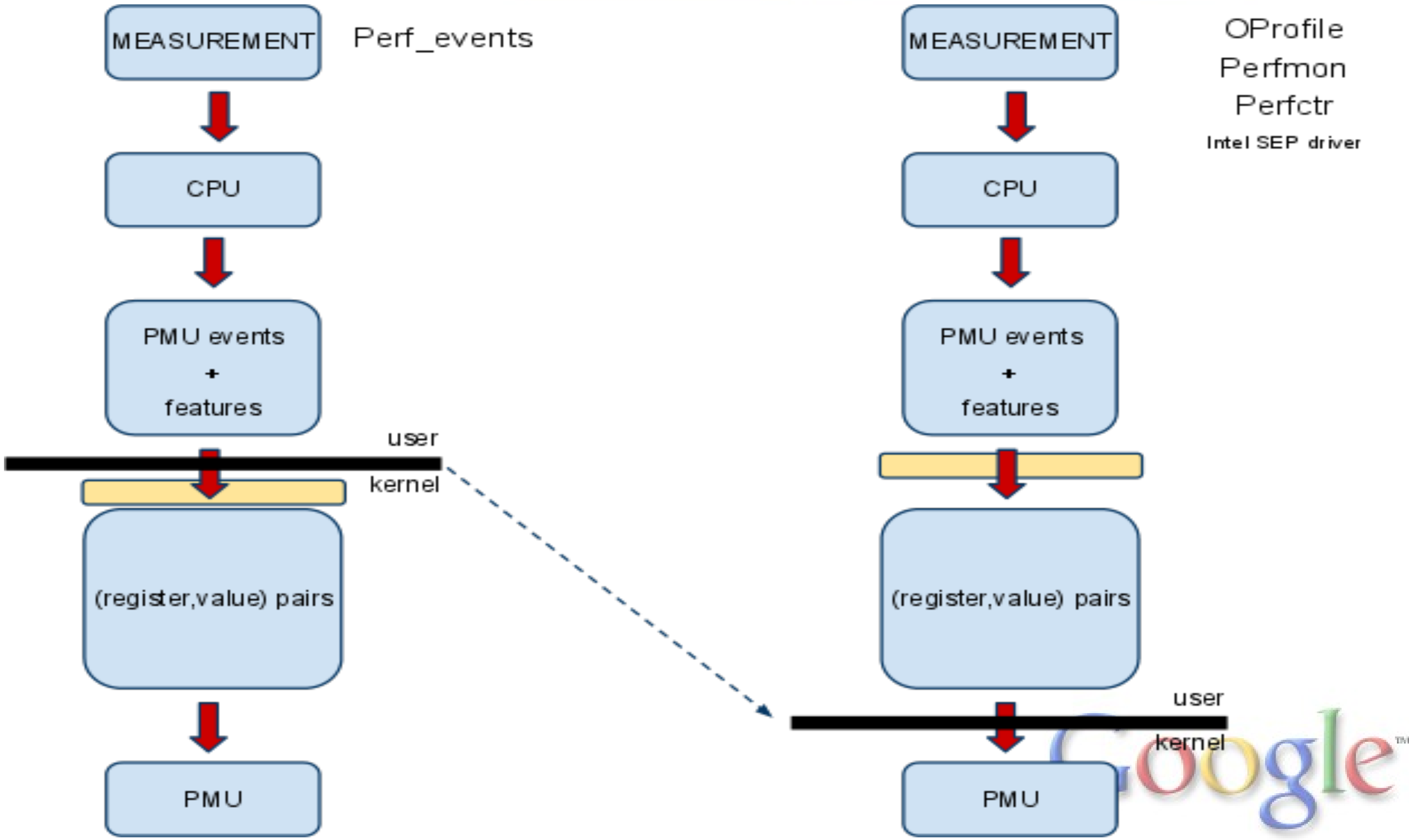
- supports per-thread and per-cpu monitoring
  - per-thread: state saved/restored on ctxsw
  - per-cpu: logical CPU, state persists across ctxsw

- supports counting and sampling
  - saves samples in a kernel buffer

- generic event-oriented API
  - not limited to PMU events
  - actual HW registers never exposed to users

- manages events independently of each other
  - event identified by file descriptor
  - no notion of a session
- available since 2.6.31(ABI changed in 2.6.32)

event vs. register oriented API



Perf_event system calls (1)

- adds *"one"* system call to setup an event
  - get a file descriptor back to identify event
  - normal file sharing semantics apply

```
int perf_event_open(struct perf_event_attr *hw,
                    pid_t pid,
                    int cpu,
                    int grp,
                    int flags)
```

hw	describes event and sampling configuration
pid	target thread, 0=self, -1=cpu-wide mode
cpu	CPU to monitor (can be used in per-thread mode)
flags	provision to extend the number of parameters
grp	file descriptor of group leader event

Perf_event system calls (2)

- counts extracted via read()
    - counts are 64-bit wide (64-bit emulation)
    - also returns scaling infos

- terminate event via close()

- additional commands via ioctl()
    - enable, disable, reset, rewrite period, refresh, filter, output

- prctl(PERF_EVENT_ENABLE/PERF_EVENT_DISABLE)
    - only works on events created by calling thread

- kernel event buffer mapping via mmap()

Events

- events have types:
  - hardware: generic PMU events
  - software: page faults, context switches, ...
  - tracepoint: kernel trace points
  - hw_cache: generic cache events (cache, TLB, BPU)
  - raw: actual PMU events
  - hw breakpoints: arbitrary data/code breakpoints

- generic PMU events:
  - mimic Intel architected PMU
  - mapped to actual PMU events by kernel
  - lack precise definitions: what is actually measured?

Generic hardware events

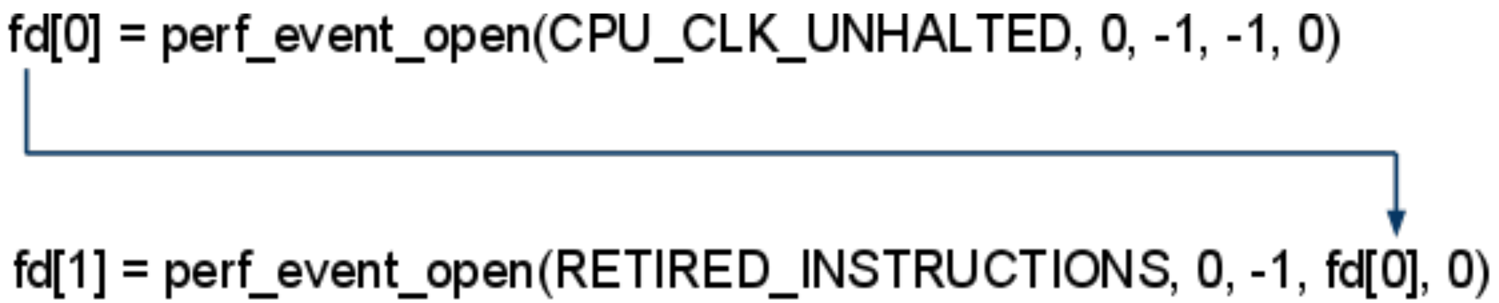
PERF_COUNT_HW_CPU_CYCLES	no precise definition yet
PERF_COUNT_HW_INSTRUCTIONS	no precise definition yet
PERF_COUNT_HW_CACHE_REFERENCES	no precise definition yet
PERF_COUNT_HW_CACHE_MISSES	no precise definition yet
PERF_COUNT_HW_BRANCH_INSTRUCTIONS	no precise definition yet
PERF_COUNT_HW_BRANCH_MISSES	no precise definition yet
PERF_COUNT_HW_BUS_CYCLES	no precise definition yet

- mapping to actual HW events not exposed

Event grouping

- events are independently scheduled on PMU
  - reliable event ratios => events must measure at the same time

- event group:
  - events are guaranteed to be scheduled together
  - cannot have more events than counters
  - created by chaining file descriptors



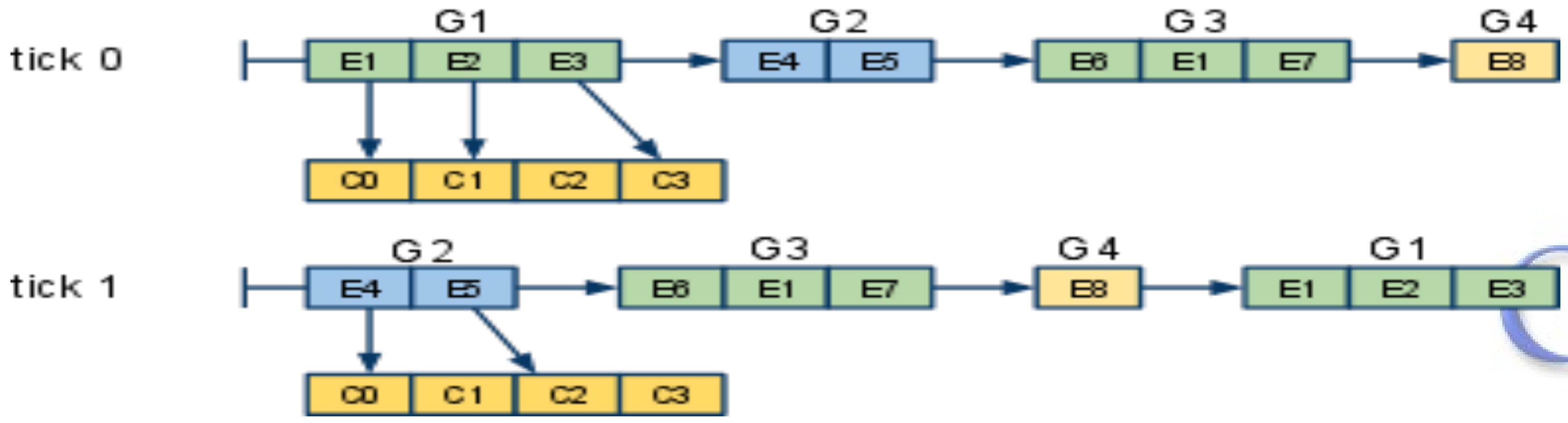
```
fd[0] = perf_event_open(CPU_CLK_UNHALTED, 0, -1, -1, 0)

fd[1] = perf_event_open(RETIRED_INSTRUCTIONS, 0, -1, fd[0], 0)
```

Events scheduling

- event groups scheduled on
  - timer tick (issue w/ tickless per-cpu), start, ctxsw
  - multiplexing when PMU is overcommitted (each tick)
  - 2.6.3[34] correct scheduling for X86 processor by Google

- group round-robin list rotated each timer tick
  - scheduling guaranteed for list head
  - stop at 1st error => bad => not maximizing PMU usage
  - stop at 1st error => good => algorithm bound by #cntrs



more event scheduling

- maximize PMU usage
    - to provide better quality counts

- maximize by scanning the whole event list
    - pros:
        - fill up the PMU
    - cons:
        - unbounded algorithm (list can be very large)
        - selection bias towards smaller groups or groups with fewer event constraints

avoiding selection bias

- discussed on LKML (http://lkml.org/lkml/2010/5/7/132)

- ensure fair scheduling of events
  - if 3 groups, then each should get 1/3rd of the time
  - regardless of constraints
- Zijlstra's algorithm:
  - each event $E(i)$ keeps its time running on CPU $s(i)$
  - schedule $E(i)$ if $s(i) < avg(s(j))$ for all j
  - stop at 1st schedule failure
  - problems: needs sort algorithm

```
evts A B C
s(0) 0 0 0 -> avg = 0/3=0.00, sort = A, B, C, schedule A, B
s(1) 1 1 0 -> avg = 2/3=0.66, sort = C, A, B, schedule C (A, B > avg)
s(2) 1 1 1 -> avg = 3/3=1.00, sort = A, B, C, schedule A, B
s(3) 2 2 1 -> avg = 5/3=1.66, sort = C, A, B, schedule C (A, B > avg)
s(4) 2 2 2 -> avg = 6/3=2.00, sort = B, C, A, schedule B, C
s(5) 2 3 3 -> avg = 8/3=2.66, sort = A, B, C, schedule A (B, C > avg)
s(6) 3 3 3 -> avg = 9/3=3.00, sort = A, B, C, schedule A, B
```

Per-thread vs per-cpu priority

- concurrent per-thread and per-cpu events supported

- pinned event
  - no multiplexing (but counter assignment can change)
  - can share PMU with other groups
  - example: NMI watchdog using perf_events (2.6.35)

- flexible event
  - can be multiplexed

- scheduling priority:
  1. per-cpu pinned events
  2. per-thread pinned events
  3. per-cpu flexible events
  4. per-thread flexible events

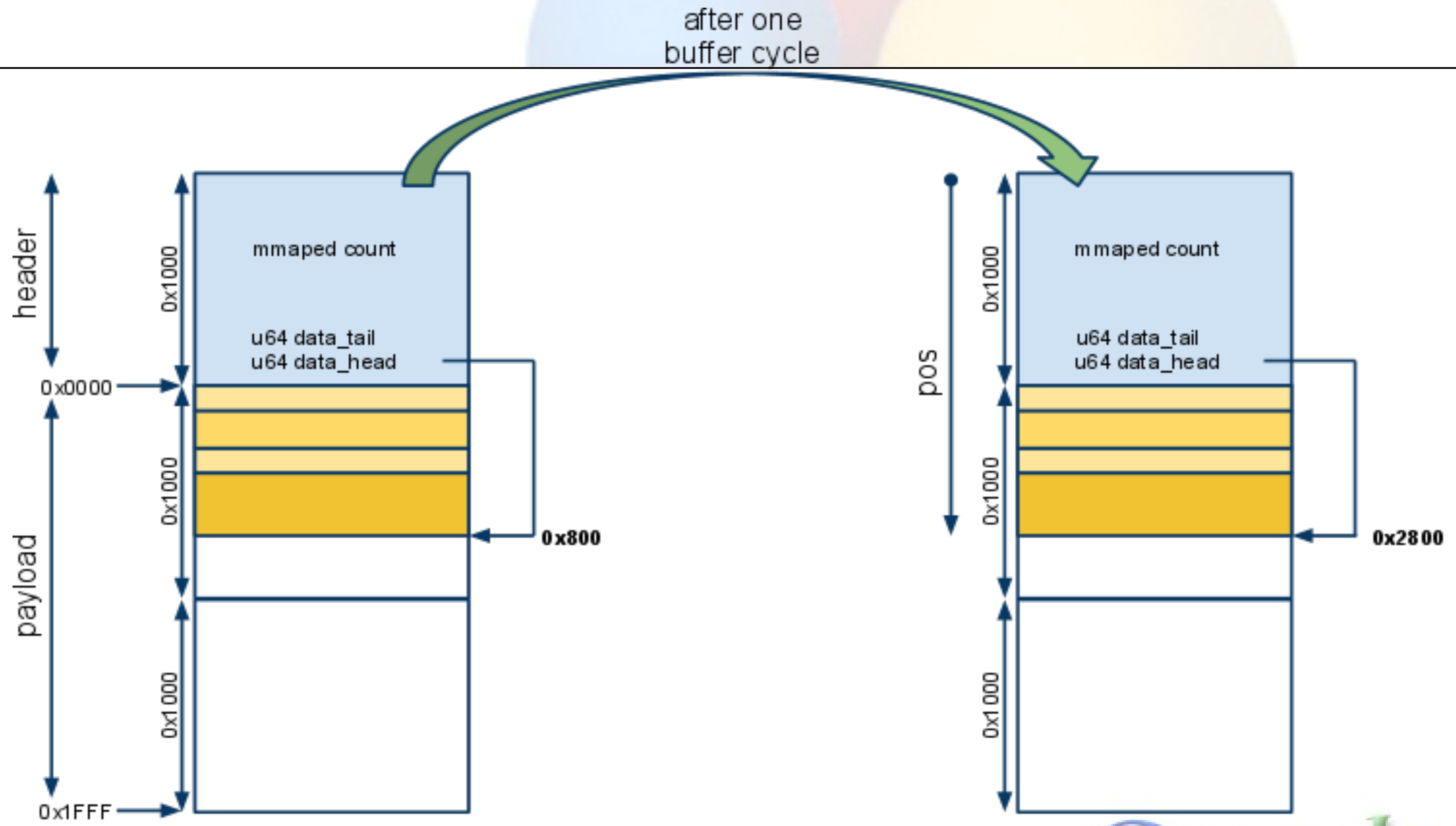
sampling buffer

- samples saved in kernel buffer
  - size determined via mmap(): 1+2^n pages
  - one buffer per event or group
  - event buffer sharing via ioctl(PERF_COUNTER_IOC_SET_OUTPUT)

- buffer format
  - fixed size header: head/tail pointers (first page)
  - universal sample: variable-size (type,size)
  - can record more than just PMU events

- cyclic read-write buffer
  - use head + tail pointers, stop if head == tail
  - can lose events: LOST event type
  - buffer cycle detection possible via data_head

sampling buffer pointers



pos = getpagesize() + data_head & (getpagesize()*(nr_pages-1)-1)

buffer payload size must be power of 2

sampling periods

- supports 64-bit sampling periods
  - read() on sampling event = accumulated counts

- sampling interval: number of occurrences of event
  - every 2000 LLC_MISSES (event-based sampling)

- sampling interval: average sampling rate (Hz)
  - e.g., LLC_MISSES at 1000Hz (1000 samples/s)
  - kernel adjusts period each tick to achieve desired Hz
  - updated period logged in sampling buffer

- X86 using NMI for PMU interrupt
  - can collect samples inside kernel's critical sections

average target rate implementation

- at each timer tick:

```
d = current_event_count - prev_event_count
prev_event_count = current_event_count
d = (d+7)/8 /* correction factor */
p = d * ticks_per_sec / target_rate_in_hz
```

- rate mode => time-based sampling
  - bias towards sections of code that run longer even though sampling event occurrence rate is identical?
  - or just a profile interpretation problem?

- rate mode is default mode for perf tool

rate vs. period interpretation example

- 10 misses/100 instr
- 2 phases (same number of instr):
  - phase 1: 1e9 instr/s for 10s
  - phase 2: 2e9 instr/s for 5s
- target rate 1000 cache miss samples/s (1000Hz)

- rate base mode:
  - phase 1: 1e8 misses/s => period = 1e5 = 10000 samples
  - phase 2: 2e8 misses/s => period = 2e5 = 5000 samples

- fixed period mode:
  - assume period = 1000 misses/sample
  - phase 1:10s@1e8 misses/s = 1e9 misses = 1e6 samples
  - phase 2: 5s@2e8 misses/s = 1e9 misses = 1e6 samples

sampling period randomization

- not support yet
- Google proposed a patch:
  - added random_width config option
  - vary p +/- random_width/2 => average is p
- patch rejected
  - no support for randomization in sampling rate mode
  - no use case in perf tool
- is that necessary?
  - not clear what it buys you? what are you measuring?

Intel LBR support

- records taken branch trace (src, dst)
  - cyclic buffer hosted in registers
  - can freeze on PMU interrupt

- LBR useful for:
  - basic block profiling
  - statistical call graph
  - path that lead to a cache miss

- used internally to correct PEBS off by 1 issue

- Google proposed patch to expose LBR to user:
  - PERF_SAMPLE_BRANCH_STACK
  - content: nr branches, then { flags, src, dst }/branch
  - rejected because missing use case in perf tool

Intel PEBS support

- captures machine state at retirement of instr. which caused an occurrence of the sampling event
  - sample stored in memory buffer
  - limited to certain events

- initial support in 2.6.35
  - exports instruction address (not the full machine state)

- PEBS not explicitly exposed:
  - user requests precise sampling mode

- precise sampling on Intel processors (w/ PEBS):
  - PEBS buffer setup for 1 sample/interrupt
  - off by 1 IP corrected using LBR, if possible (precise>=2)
  - corrected sample marked with PERF_RECORD_MISC_EXACT_IP

Supported HW

- AMD64
  - K8, Barcelona, Shanghai, Istanbul (Magny-Cours?)

- Intel X86
  - P6, Core Duo/Solo, Netburst (P4)(2.6.35)
  - Atom, Core, Nehalem/Westmere
  - any processor with architected perfmon (PMU)

- ARM
  - ARMV6 (1136,1156,1176)
  - ARMV7 (cortex-a8, cortex a9)

- IBM Power

still missing

- Intel Nehalem: OFFCORE_RESPONSE_*
  - use extra MSR (shared when HT is on)
- Intel Nehalem: LBR_SELECT
  - LBR filtering (shared when HT is on)
  - likely no support
- Intel Nehalem uncore (Intel contributing)
  - internal restructuring to support distinct PMUs
  - PMU naming scheme (likely sysfs)
- AMD IBS (AMD contributing)
  - patch proposed by AMD 6 months ago
  - not yet in, may be revisited by sysfs restructuring

perf tool

- included in kernel source tree (tools/perf)

- support for per-thread, per-cpu counting, sampling, tracing
  - cmdline tool, curses-based gui
  - operate like git: perf command arguments

- profiling support similar to Oprofile
  - collection => perf record => binary output file
  - high level analysis => perf report
  - source level analysis => perf annotate
  - remote collect, local analysis possible
- top-like mode => perf top

libpfm4

- helper library to map event names to event encoding
  - rewritten from scratch for perf_events
  - core code independent of OS API

- improved event string
  - ex: INST_RETIRED:ANY_P:c=1:i:u:k

- provide OS specific API to initialize syscall structures
  - pfm_get_perf_event_encoding(char *, struct perf_event_attr *)

- HW support
  - all X86 processors, IBM Power

- git repository: perfmon2.sf.net

Conclusion

- signal problem solved
- perf_events has improved
  - more HW support
  - better event scheduling
- perf_events is still missing a some key HW features
- perf_event tool available: perf
- libpfm4 provides perf_events support