Linux new monitoring interface: Performance Counter for Linux

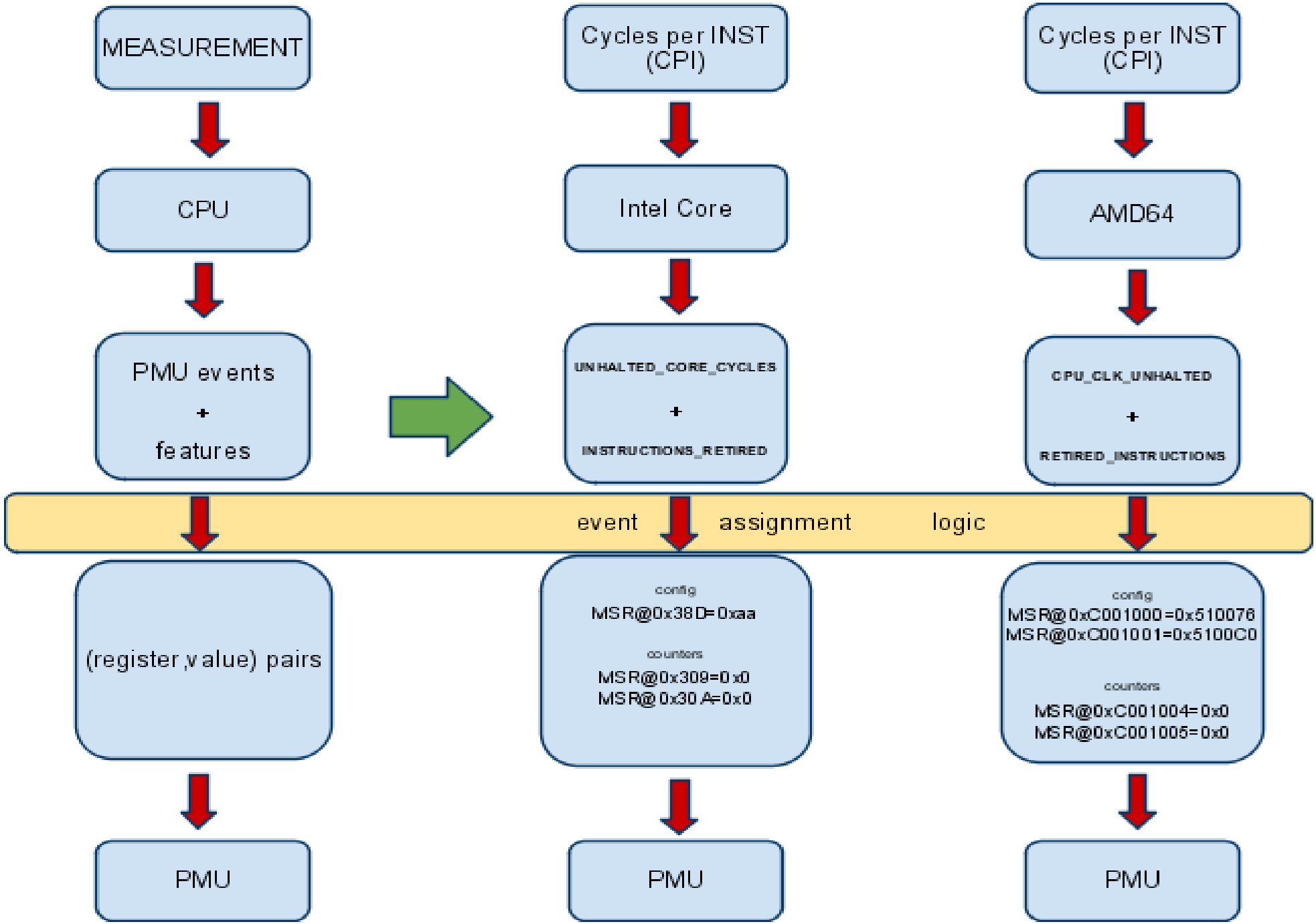
CSCADS workshop, Lake Tahoe July 2009

eranian@google.com

Where does PCL come from?

- counter-proposal to perfmon
  - as perfmon was introduced into linux-next (very late)
- created by Molnar, Zijlstra, Gleixner (Dec'08)
  - all Linux x86 maintainers
  - strong x86 influence
  - ported to Power rapidly by McKerras (Linux PPC)
- code included in Linus's 2.6.31 kernel
  - examples in `tools/perf_counters`
  - documentation in `tools/perf_counters`

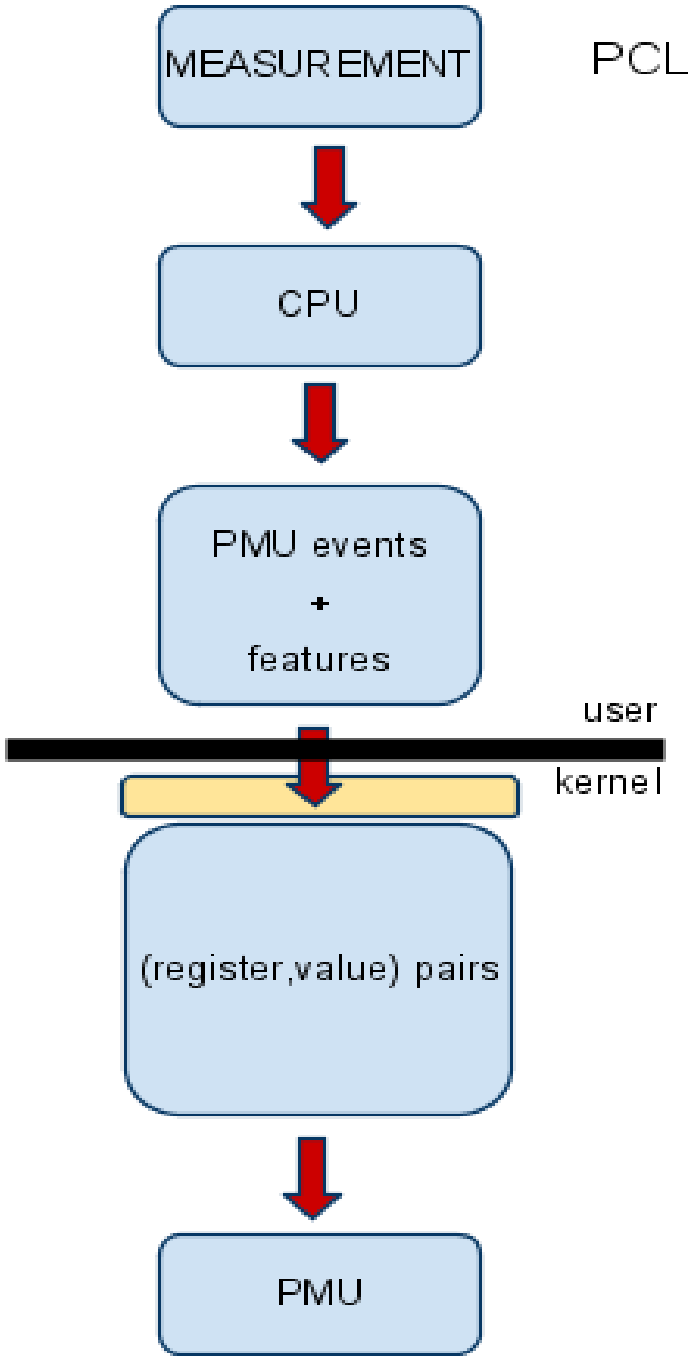
measurement breakdown

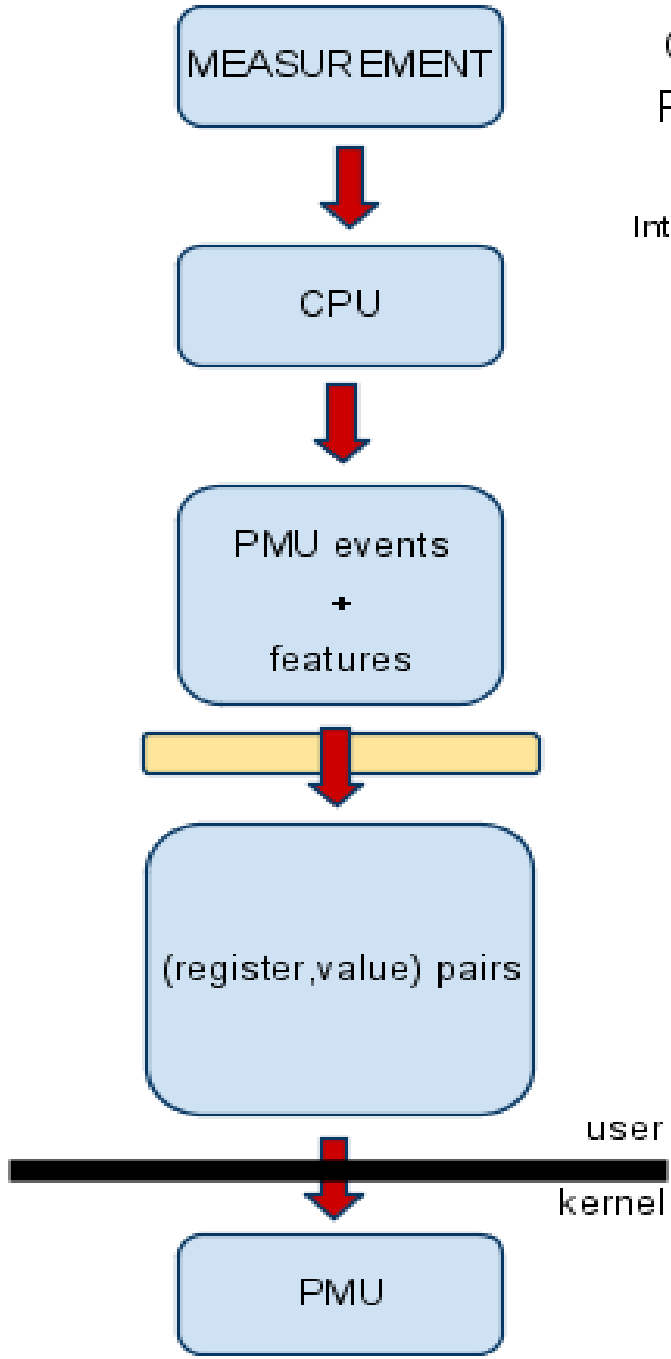


PCL key design choices

- supports per-thread and cpu-wide monitoring
  - per-thread: state saved/restored on ctxsw
  - cpu-wide: logical CPU, state persists across ctxsw
- supports counting and sampling
  - save samples in a kernel buffer
- generic event-oriented API
  - not limited to PMU events
  - never expose actual hardware resource to users
- manages events independently of each other
  - event identified by file descriptor
  - no notion of a session (events + target thread or CPU)
- system call oriented API (not a driver)

event vs. register oriented API (1)





event vs. register oriented API (2)

- event-oriented
  - pros:
    - quick ramping up period (read Intel Appendix A)
    - mapping event -> register can change dynamically
  - cons:
    - event -> assignment logic pushed into kernel
    - create an abstraction to expose non-counting events

- register-oriented
  - pros:
    - simpler kernel, easier maintenance
    - more error-prone code in user-land
  - cons
    - harder to change assignment dynamically
    - more difficult to expose non register style features

PCL 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_counter_open(struct perf_counter_attr*hw,
                      pid_t pid,
                      int cpu,
                      int grp,
                      int flags)
```

`hw`	describes event and sampling configuration (64-byte struct)
`pid`	target thread, 0=self, -1=cpu-wide mode
`cpu`	CPU to monitor, -1=per-thread mode
`flags`	provision to extend the number of parameters
`grp`	used to create groups

PCL perf_counter_attr structure

```
struct perf_counter_attr {
        __u32                   type;
        __u32                   size;
        __u64                   config;
        union {
                __u64           sample_period;
                __u64           sample_freq;
        };
        __u64                   sample_type;
        __u64                   read_format;

        __u64                   disabled       :  1,

                                inherit        :  1,
                                pinned         :  1,
                                exclusive      :  1,
                                exclude_user   :  1,
                                exclude_kernel :  1,
                                exclude_hv     :  1,
                                exclude_idle   :  1,
                                mmap           :  1,
                                comm           :  1,
                                freq           :  1,
                                inherit_stat   :  1,
                                enable_on_exec :  1,
                                __reserved_1   : 51;
        __u32                   wakeup_events;
        __u32                   __reserved_2;
        __u64                   __reserved_3;
        __u64                   __reserved_4;
};
```

PCL system calls (2)

- counts extracted via `read()`
  - +multiplexing timing infos, +sampling identifier
  - counts are 64-bit wide (64-bit emulation)
- termination via `close()`
- additional commands via `ioctl()`
  - enable, disable, reset, rewrite period, refresh
- kernel event buffer mapping via `mmap()`
- sample notification via `fcntl(O_ASYNC/F_SETOWN)`

effectively a total of 6 system calls

PCL events

- events have types (defined as enum):
  - `hardware`: used for generic PMU events
  - `software`: page faults, context switches, ...
  - `tracepoint: ??`
  - `hw_cache:` generic cache events (cache, TLB, BPU)
  - `raw`: actual PMU events
  - more needed: uncore PMU, chipset counters

- generic PMU events (defined as enum):
  - mimic Intel architected PMU
  - mapped to actual PMU events by kernel
  - lack precise definitions: what do they measure?

PCL generic hardware events

PERF_COUNT_CPU_CYCLES	no precise definition yet
PERF_COUNT_INSTRUCTIONS	no precise definition yet
PERF_COUNT_CACHE_REFERENCES	no precise definition yet
PERF_COUNT_CACHE_MISSES	no precise definition yet
PERF_COUNT_BRANCH_INSTRUCTIONS	no precise definition yet
PERF_COUNT_BRANCH_MISSES	no precise definition yet
PERF_COUNT_BUS_CYCLES	no precise definition yet

PCL software events

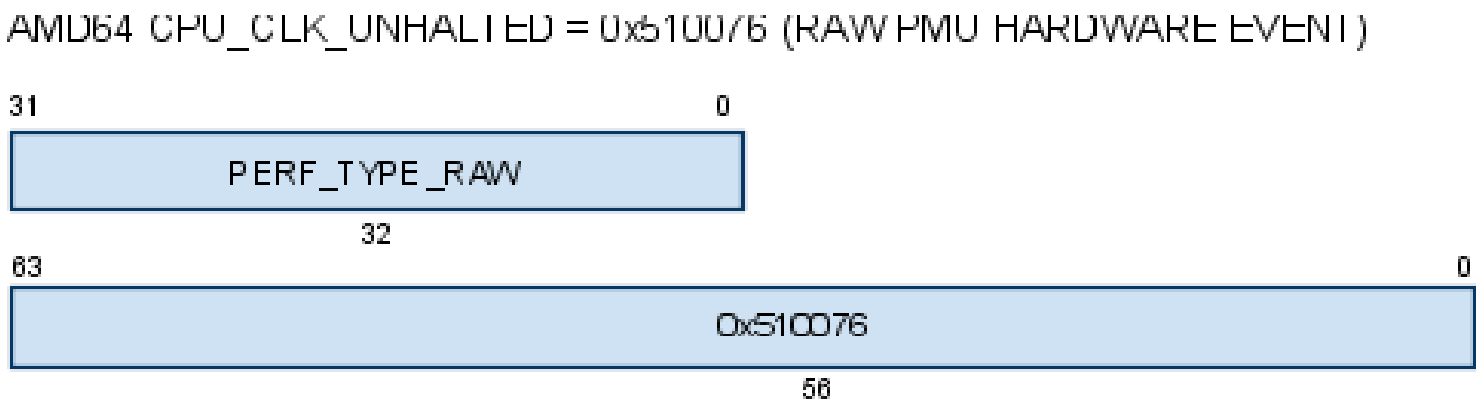
PERF_COUNT_CPU_CLOCK	wall-clock time
PERF_COUNT_TASK_CLOCK	virtual time
PERF_COUNT_PAGE_FAULTS	page faults
PERF_COUNT_CONTEXT_SWITCHES	context switches out of monitored task
PERF_COUNT_CPU_MIGRATIONS	task migrations
PERF_COUNT_PAGE_FAULTS_MIN	minor page faults
PERF_COUNT_PAGE_FAULTS_MAJ	major page faults

PCL generalized cache events

- provide generic events for common cache metrics
  - mapped onto actual PMU event if exist
- covers 3-dimensions:
  - { L1-D, L1-I, LLC, ITLB, DTLB, BPU }
  - { read, write, prefetch }
  - { accesses, misses }
  - examples: L1D.read.misses
- no precise definitions exist for generic events
- false good idea
  - subtle differences make events difficult to compare

PCL event encoding

- event encoded as `uint64_t`
  - if code > 64-bits, can use a reserved field
- privilege levels expressed via dedicated fields
  - `exclude_user, exclude_kernel, exclude_hv`
  - overrides priv level in the raw event code

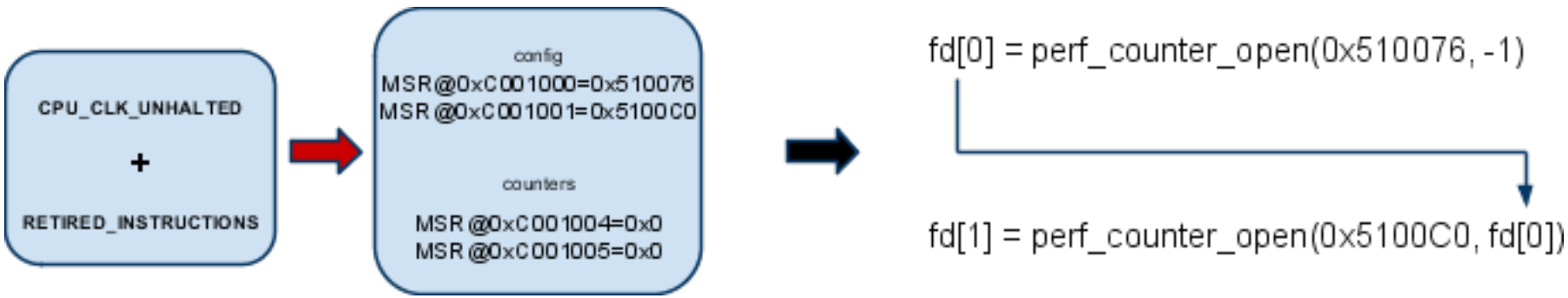


PCL event grouping

- events are independently scheduled on PMU
  - reliable event ratios => guarantee on events scheduling

- PCL event group
  - events are guaranteed scheduled together
  - #events <= #counters
  - created by chaining file descriptors
  - 1st event = group leader

- start/stop group via group leader
  - groups can be scheduled if all its events are enabled
  - cannot read all counts via `read()` on group leader



PCL event assignment logic

- only performed by kernel

- code is PMU or architecture specific
  - fairly trivial on X86 (both Intel and AMD)
  - very difficult on Itanium for certain events
  - may need global view of all events sharing the PMU

- assignment performed on every counter activation:
  - activation: ctxsw in, multiplexing in, counter start
  - necessary because of PMU sharing
  - lazy approach to mitigate cost: try to reuse previous reg

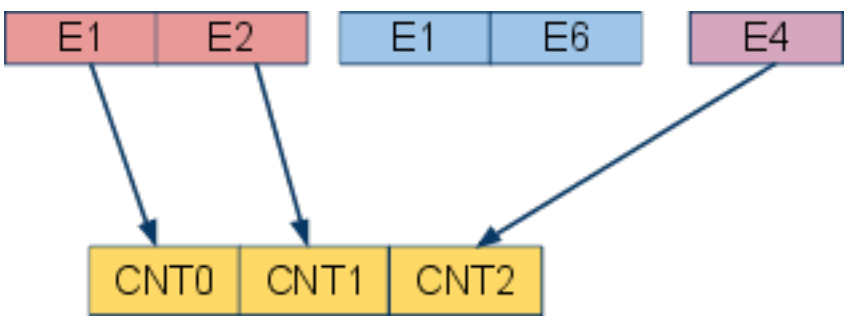
- if kernel wrong => kernel patch
  - kernel.org release cycle != distro release cycle
  - no user bypass exists

PCL PMU sharing (1)

- per-thread and cpu-wide can run concurrently
- multiple tools may be monitoring the same thread or CPU

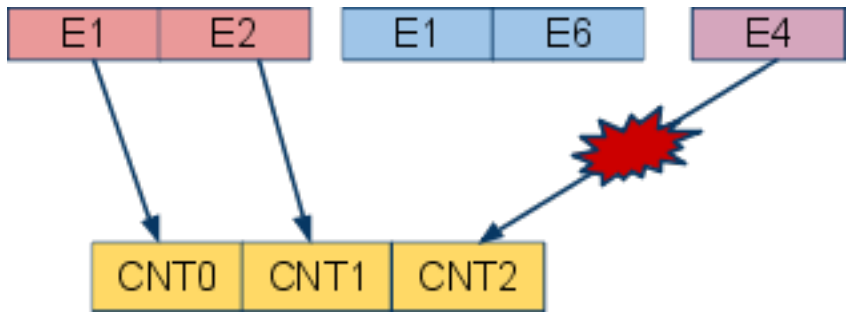
- event groups are independently scheduled on PMU

- PMU is shared between event groups by default
  - groups may come from different tools/users
  - kernel must ensure groups are compatible (no leaks)
  - no back-to-back group scheduling guarantee



PCL PMU sharing(2)

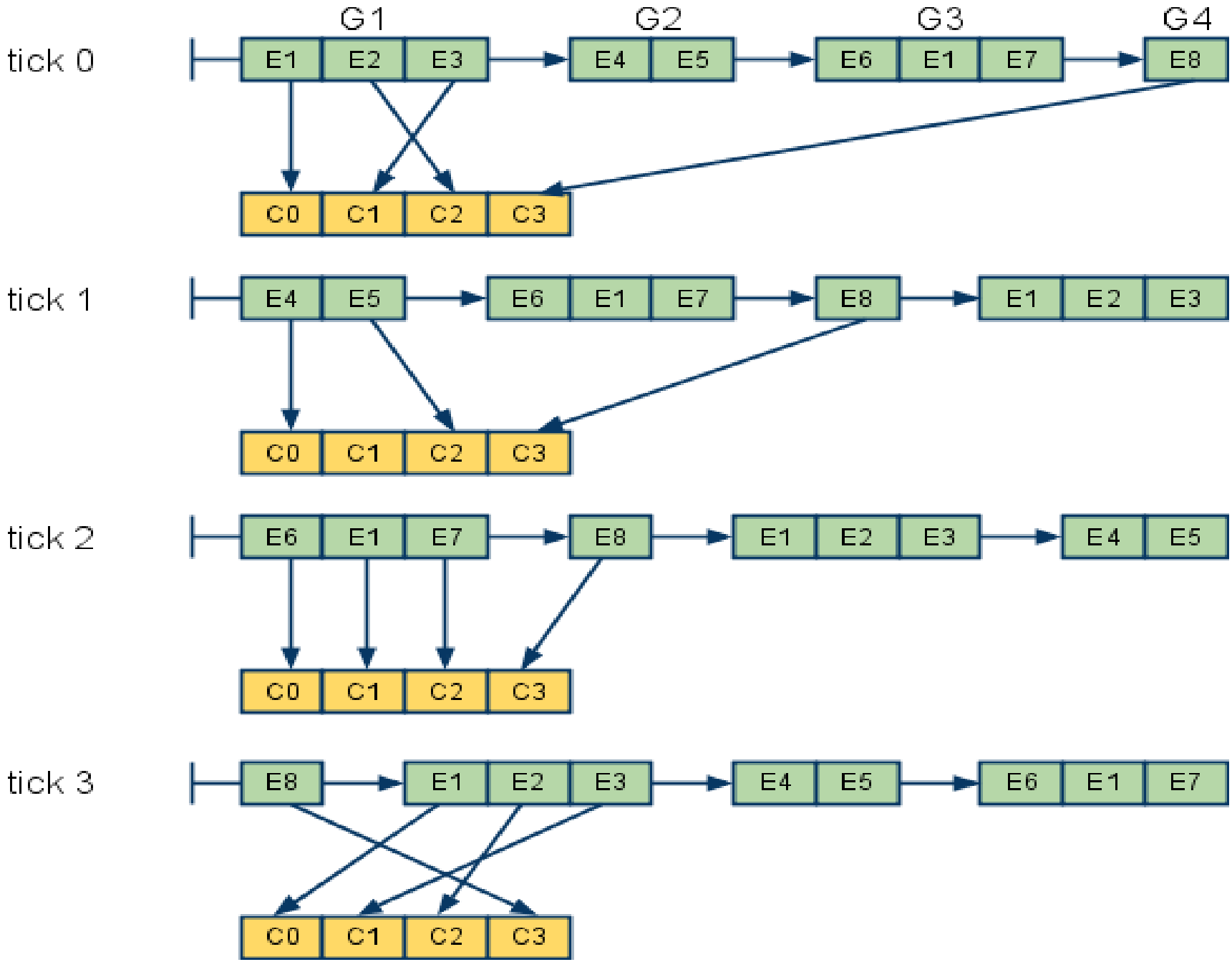
- `exclusive` mode:
  - only this group can use the PMU
  - does not prevent multiplexing, just sharing
  - can be combined with pinning



PCL group multiplexing

- multiplex events to allow sharing when overcommitted
- multiplexing occurs by default
  - group granularity
  - order: group creation (no back-to-back guarantee)
- time-based multiplexing only
  - switch timeout = 1 tick (must disable tickless in syswide)
  - not controllable at this point
- scaling of counts at the user level
  - time tracking enabled via `read_format`
  - `read()` returns: count, total_time, time_active
- can prevent multiplexing via `pinned`
  - group stays on PMU until stopped

PCL group scheduling: selection bias?



PCL mmap'd counts

- avoid cost of system call to read count for self-monitoring
  - `getpid()` = 500 cycles (Q6600 2.4GHz)
  - `read(count)` = 2700 cycles (Q6600 2.4GHz)

- leverage HW ability to read registers from user level
  - rdpmc(50 cycles) vs. rdmsr (226 cycles) (Q6600 2.4GHz)

- `mmap()` SW counter + recombine with HW counter
  - uses 1 page/event (pressure on `RLIMIT_MEMLOCK`)
  - timing to scale count exposed to support multiplexing

```
do {
     barrier();prev_lock = mmap->lock
     if (mmap->index)
         count = rdpmc(mmap->index -1);
     else
         goto regular_read_syscall;
     count += mmap->offset;barrier()
} while (prev_lock != mmap->lock);
```

PCL sampling periods

- PCL has notion of a sampling period (!= counter value)
  - sampling period is 64-bit wide
- support for event-based sampling
  - period = #occurrences (e.g., 2000 `LLC_MISSES`)
  - sampling on SW events possible
- sampling interval can be expressed as frequency
  - kernel adjusts period each tick to achieve desired Hz
  - updated period logged in event buffer
- no period randomization yet

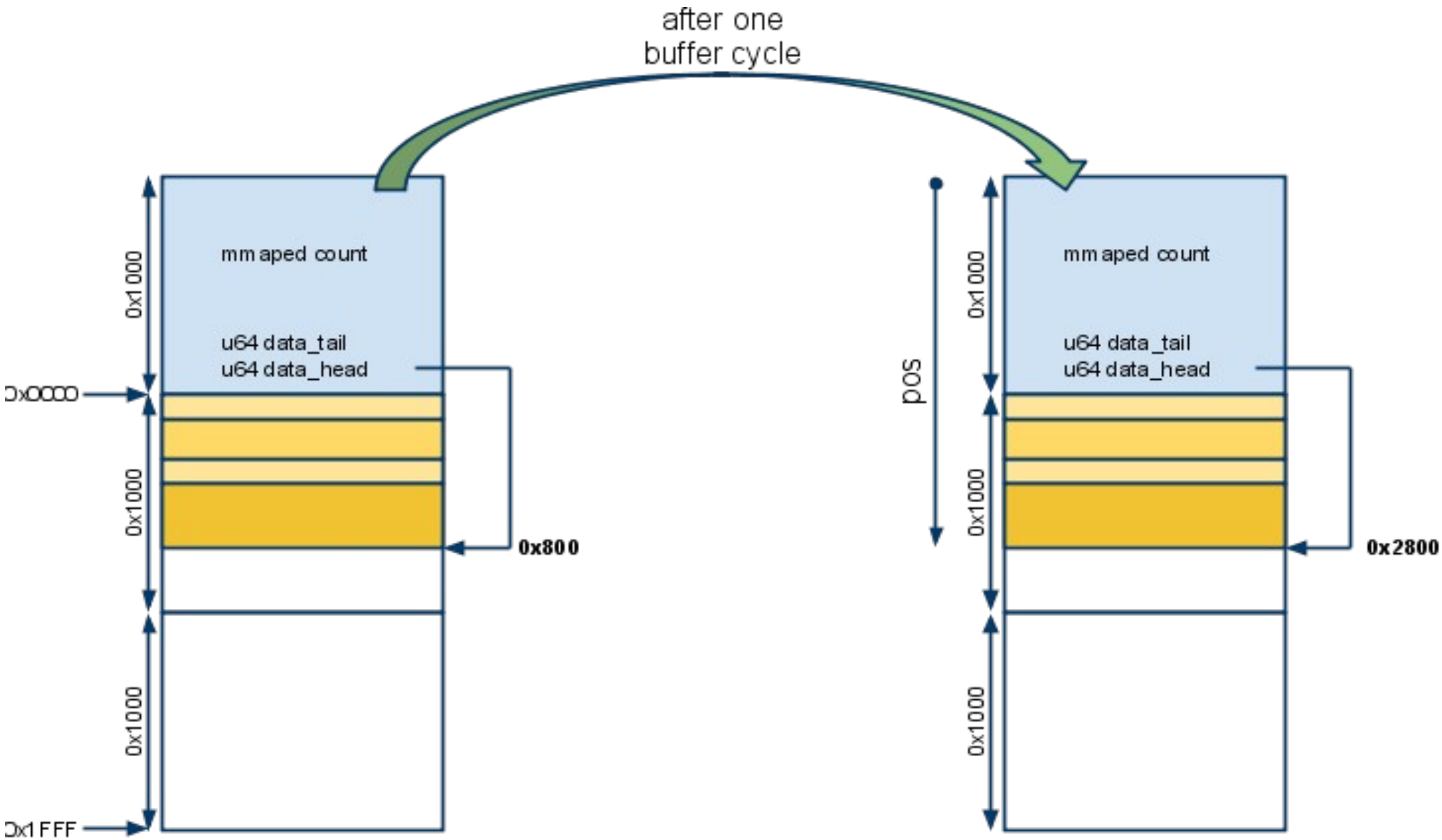
PCL sampling buffer

- samples saved in kernel event buffer
  - size determined via `mmap()`
  - at least 2 pages
  - one buffer per event or group

- buffer format
  - fixed size header: position + mmap'd count (1 page)
  - universal sample: variable-size (`type,size`)
  - can record more than just PMU events

- cyclic read-write buffer
  - when buffer full, wait for user notification via write to hdr
  - current offset via `data_head` index in header
  - buffer cycle detection possible via `data_head`
  - can have lost samples: LOST event type

PCL sampling buffer positioning



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

PCL sampling notifications

- event notification on buffer page crossing
  - 2-page = double-buffer (notify halfway)
  - no control over notification point
- multiple events can be written concurrently
  - events appear (`data_head`) in order to user
- support for `poll()/select()`
- asynchronous notification via `SIGIO`
  - required for self-sampling
  - signal handler nesting possible

PCL sample attributes

- fixed size sample header: `{ type, misc, size }`
  - variable-size body

- `sample_type` bitmask to select what to record/event
  - layout: order of increasing enum value

`PERF_SAMPLE_IP`	interrupted code address
`PERF_SAMPLE_TID`	PID, TID
`PERF_SAMPLE_TIME`	`sched_clock()`
`PERF_SAMPLE_ADDR`	extra 64-bit address??
`PERF_SAMPLE_GROUP`	values of other events in the group
`PERF_SAMPLE_CALLCHAIN`	call stack (kernel OR user)
`PERF_SAMPLE_CONFIG`	event encoding
`PERF_SAMPLE_CPU`	current CPU at time of intr
`PERF_SAMPLE_PERIOD`	last sampling period

PCL event buffer sample types

`PERF_EVENT_MMAP`	executable file mmapped
`PERF_EVENT_COMM`	process name was changed (`prctl()`)
`PERF_EVENT_PERIOD`	sampling period was changed (to adjust frequency)
`PERF_EVENT_THROTTLE`	event group monitoring stopped because of excessive interrupts
`PERF_EVENT_UNTHROTTLE`	event group monitoring restarted after being throttled due to excessive interrutps
`PERF_EVENT_FORK`	`fork`() event
`PERF_EVENT_LOST`	report number of events lost due to user being too slow to extract events
`PERF_EVENT_READ`	report event count a specific sites, e.g, parent -> child
`PERF_EVENT_SAMPLE`	counter generated sample

PCL PMU interrupt throttling

- prevent system breakdown with bogus sampling periods
  - on X86 using NMI
- sysadmin can set maximum threshold via sysctl()
  - `/proc/sys/perf_counter_int_limit`
  - rate is per CPU per second
- when rate is exceeded counter/group is stopped
  - throttling recorded in event buffer
- Restart on next timer tick
  - unthrottling recorded in event buffer

PCL symbolization support

- correlate sample addresses => binary/module/function
  - needed for both per-thread and system-wide
- can request per-event `mmap()` tracking
  - content: `pid, tid, addr, len, pgoff, filename`
- mmap sample recorded in event buffer
  - basis for tools to track full address space changes
  - cannot afford to lose one!

PCL inheritance

- event is inherited across `fork()/pthread_create()`
  - counts aggregated into parent
  - set `inherit` in `perf_counter_attr`
- enable/disable controls entire hierarchy
- sampling support except for group sampling
  - works only in single event sampling

PCL tools

- perf: sample tool in kernel tree (`tools/perf_counters`)
  - top mode: sys-wide kernel sampling
  - stat mode: similar to pfmon
  - report, annotate (similar to OProfile)

```
$ perf stat date
Mon Jun  8 13:41:45 CEST 2009

 Performance counter stats for 'date':

       3.132542  task clock ticks     #       0.914 CPU utilization factor
              2  context switches     #       0.001 M/sec
              1  CPU migrations       #       0.000 M/sec
            239  pagefaults           #       0.076 M/sec
        4938179  CPU cycles           #    1576.413 M/sec
        4056211  instructions         #    1294.862 M/sec
          74924  cache references     #      23.918 M/sec
           3637  cache misses         #       1.161 M/sec

 Wall-clock time elapsed:     3.426563 msecs
```

- PAPI substrate
  - libpfm adapted by IBM to support PCL (proof-of-concept)

PCL still missing...

- AMD64: IBS support
  - probably new `sample_type` + pseudo event
- Intel Core, Nehalem: PEBS support
  - probably new `sample_type` + event tag
- Intel Core i7: uncore PMU support
  - probably new event type
- Intel Core, Nehalem: LBR support
  - probably new `sample_type` + pseudo event
- X86 event constraints support
- True test will be full IA-64 support
  - opcode matching, range restrictions, BTB

PCL issues

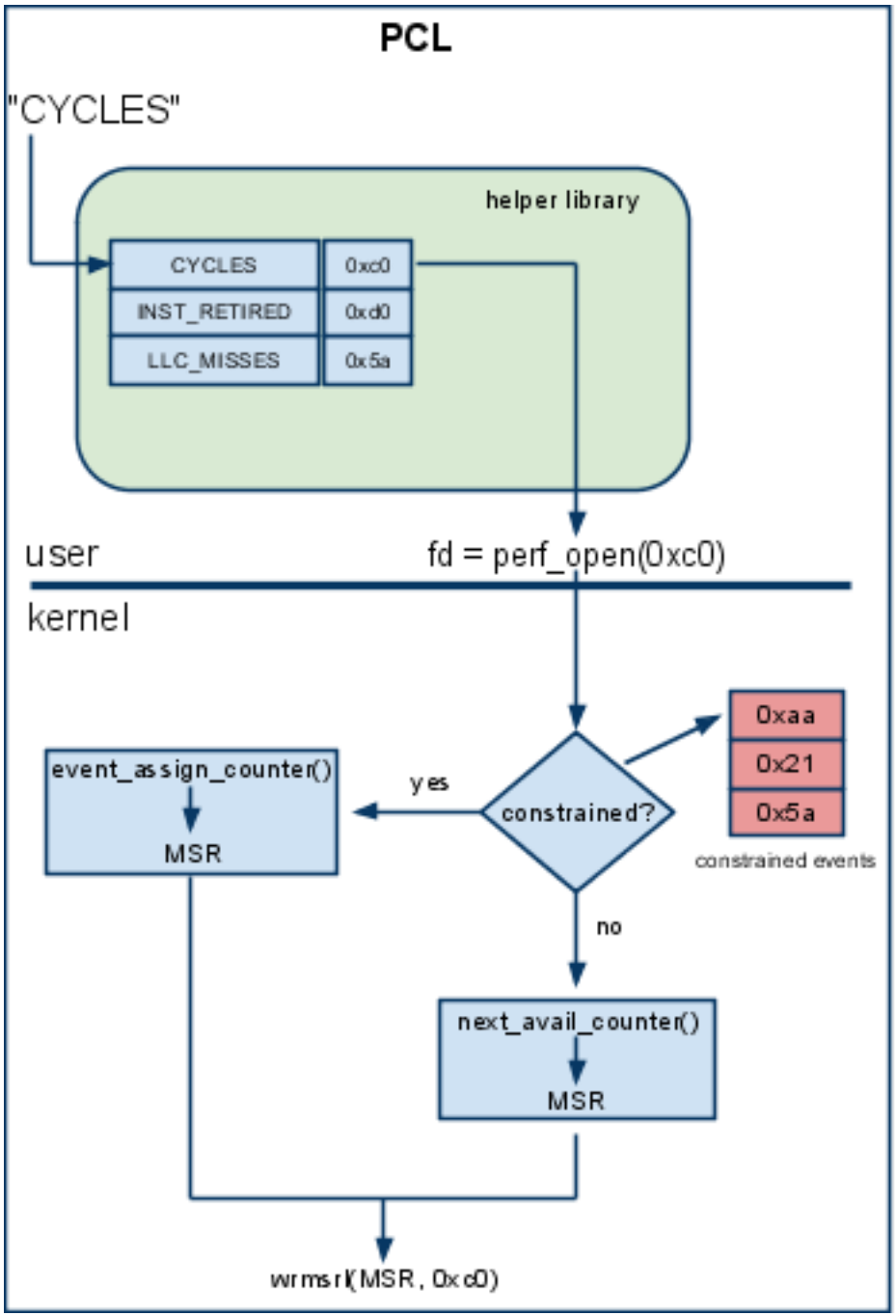
- cherry-picking features which they think are useful (to them)
  - e.g.: no motivation for Intel Core i7 uncore PMU
  - unfortunately different people have different needs
  - tool developers don't want to become kernel developers

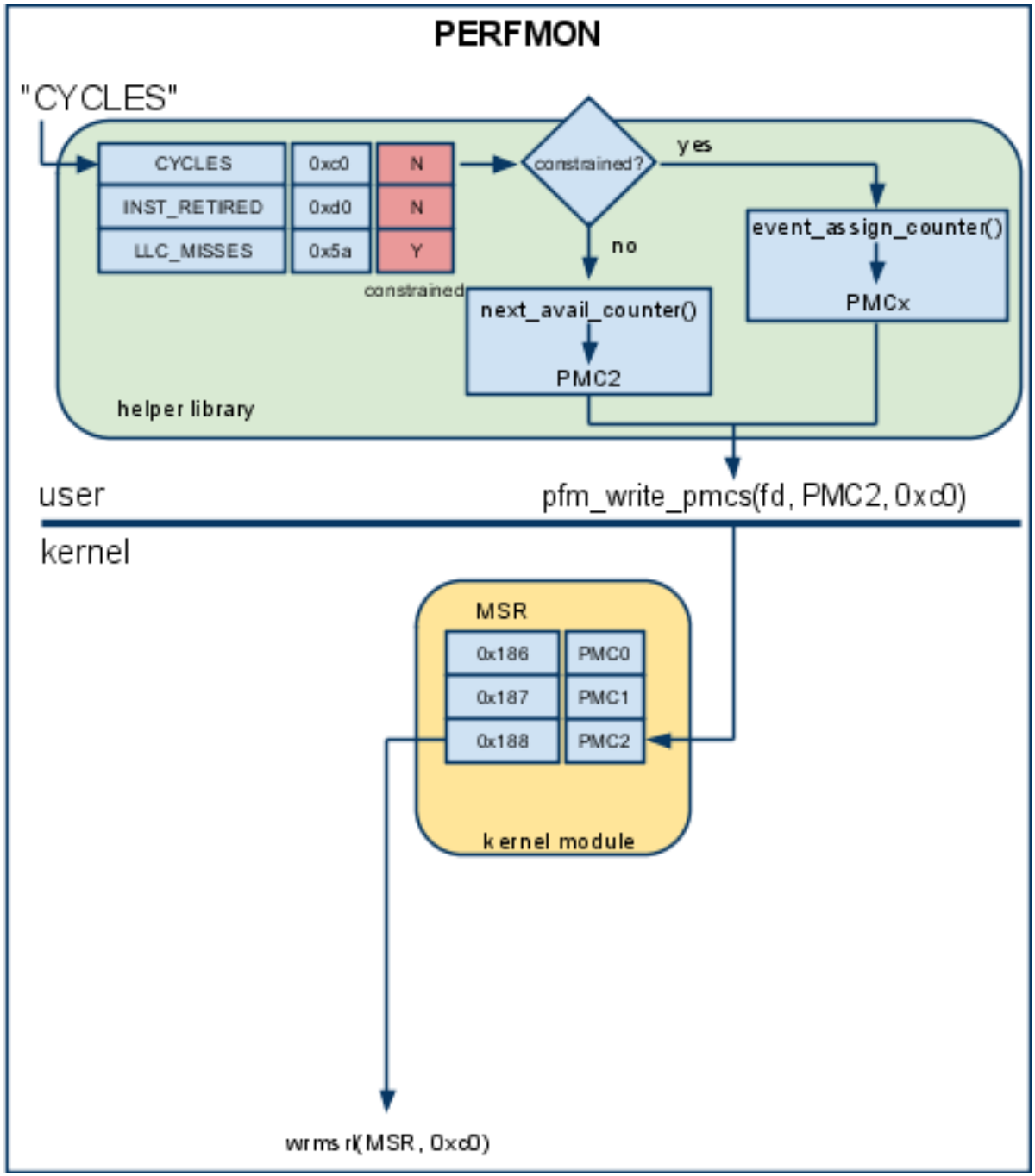
- need to address advanced features (IBS, PEBS)
  - must invent an abstraction: not too high, not too low
  - increased kernel complexity

- illusion of simplicity
  - PMU deals with micro-architecture which is complex
  - must understand what is actually measured and when

- power-user
  - need advanced features now
  - like to have full control

PCL vs. perfmon





PCL code examples

PCL self-monitor+count example

```
#include <perf_counter.h>
uint64_t val;
struct perf_counter_attr attr = {0, };
int fd;


attr.type = PERF_TYPE_HARDWARE;
attr.config = PERF_COUNT_HW_CPU_CYCLES; /* generic PMU event*/
attr.disabled = 1;


fd = perf_counter_open(&attr, getpid(), -1, -1, 0);

ioctl(fd, PERF_COUNTER_IOC_ENABLE, 0);

/* RUN CODE TO MONITOR */

ioctl(fd, PERF_COUNTER_IOC_DISABLE, 0);

read(fd, &val, sizeof(val));
printf("%"PRIu64" CYCLES\n", val);
close(fd);
```

PCL self sampling example

```
#include <perf_counter.h>

struct perf_counter_attr attr = {0,};
struct mmap_page *header;
size_t map_size;
int fd;



attr.type = PERF_TYPE_HARDWARE;
attr.config = PERF_COUNT_HW_CPU_CYCLES;
attr.sample_period = 2400000;
attr.sample_type = PERF_SAMPLE_IP;
attr.disabled = 1;

fd = perf_counter_open(&attr, getpid(), -1, -1, 0);

map_size = getpagesize() * 3;
header = mmap(NULL, map_size, PROT_READ, MAP_SHARED, fd, 0);

ioctl(fd, PERF_COUNTER_IOC_ENABLE, 0);
/* RUN CODE TO MONITOR */
```

PCL group self-sampling example

```
#include <perf_counter.h>
struct perf_counter_attr attr = { 0, };
struct mmap_page *header;
size_t map_size;
int fd[2];

attr.type = PERF_TYPE_HARDWARE;
attr.config = PERF_COUNT_HW_CPU_CYCLES;
attr.sample_period = 2400000;
attr.sample_type = PERF_SAMPLE_IP|PERF_SAMPLE_GROUP;
attr.disabled = 1;

fd[0] = perf_counter_open(&attr, getpid(), -1, -1, 0);

memset(&attr, 0, sizeof(attr));
attr.config = PERF_COUNT_INSTRUCTIONS;

fd[1] = perf_counter_open(&attr, getpid(), -1, fd[0], 0);

map_size = getpagesize() * 3;
header = mmap(NULL, map_size, PROT_READ, MAP_SHARED, fd[0], 0);
ioctl(fd[0], PERF_COUNTER_IOC_ENABLE, 0); /* start group */
/* RUN CODE TO MONITOR */
```