Performance Monitoring Unit

Team : Aman Singh, Anup Buchke

Mentor : Dr. Yann-Hang Lee

What is PMU?

- Most processors nowadays have special, on-chip hardware that monitors micro architectural events like elapsed cycles, cache hits, cache miss etc.

- It is a subsystem which helps in analyzing how an application or operating systems are performing on the processor.

Performance Monitoring Events

The Performance Monitoring Events can be broadly categorized in two types

- Hardware

Ex: CPU-Cycles, Instructions, Cache References

- Software

Ex: Page Fault, Context Switch, etc

Performance Monitoring Hardware

It consists of two components:

Performance Event Select Registers

Configuration registers to control what events to be monitored and how to monitor.

Event Counters

The registers which actually count the number of events based on the event select register’s configuration.

For monitoring an event a counter is paired with an event select register.

Intel's Performance Event Select MSR

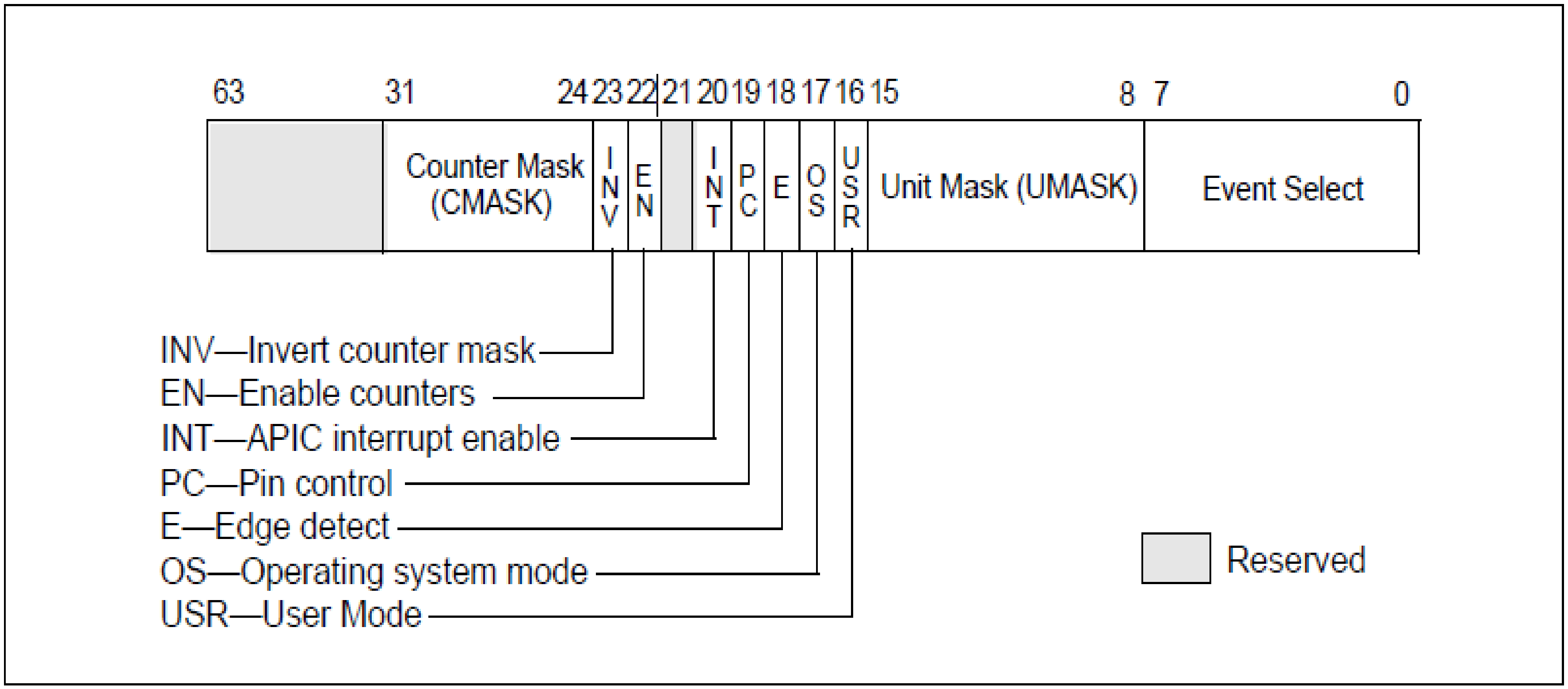


Figure 18-1. Layout of IA32_PERFEVTSELx MSRs

Important fields in the MSR

- Event Select Field : To select the event logic unit to monitor an architectural performance event.
- Unit Mask(UMASK): These bits qualify the condition that the selected event logic unit detects. So what performance event to be monitored is decided by the Event Select field plus the Unit Mask field.
- USR(User Mode) Flag: If set it means that the micro architectural condition will be counted only for privilege levels 1,2 and 3.
- OS flag(operating system mode): If set the microarchitectural condition will be counted only for privilege level 0.
- INT (APIC Interrupt enable): If set an interrupt is generated on counter overflow.

Important fields in the MSR (cond..)

- EN (Enable Counters) — When set, performance counting is enabled in the corresponding performance-monitoring counter; when clear, the corresponding counter is disabled.
- INV flag : Inverts the result of the counter mask field when set so that both greater than or less than comparisons can be made.
- CMASK : — When this field is not zero, a logical processor compares this mask to the events count of the detected microarchitectural condition during a single cycle. If the event count is greater than or equal to this mask, the counter is incremented by one. Otherwise the counter is not incremented.

Approaches for Performance Monitoring

- **Counting**

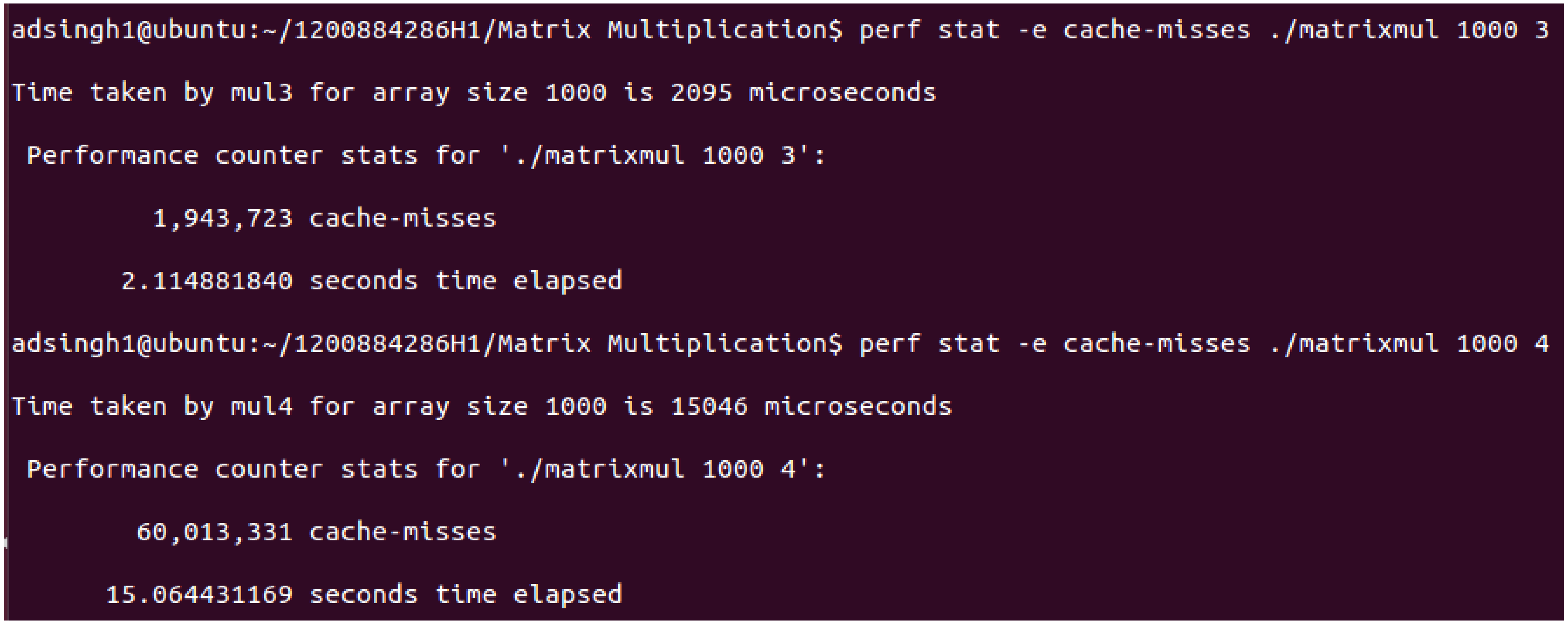
  In counting mode, the MSRs are configured before monitoring starts and at the end of monitoring period the counter values are aggregated.

- **Event Based Sampling**

  An event counter is configured to overflow after a preset number of events. At overflow the process information, like instruction pointer, general purpose registers and EFLAG registers, is captured and passed to user.

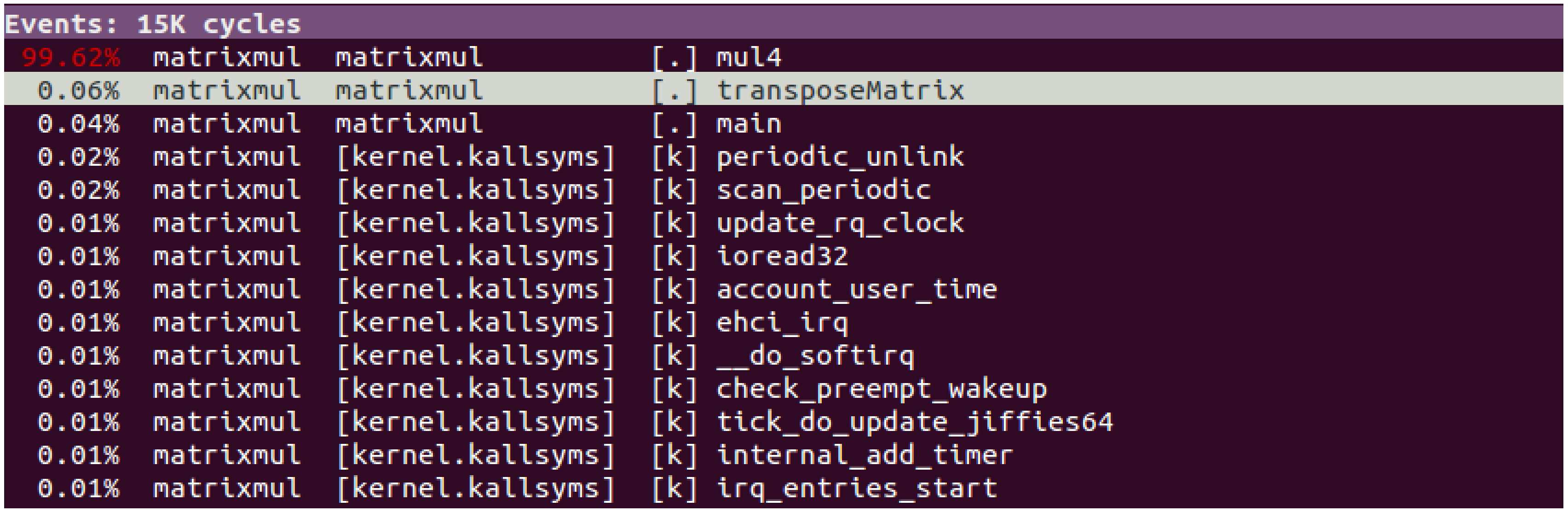
Counting Example

- Using the perf tool :



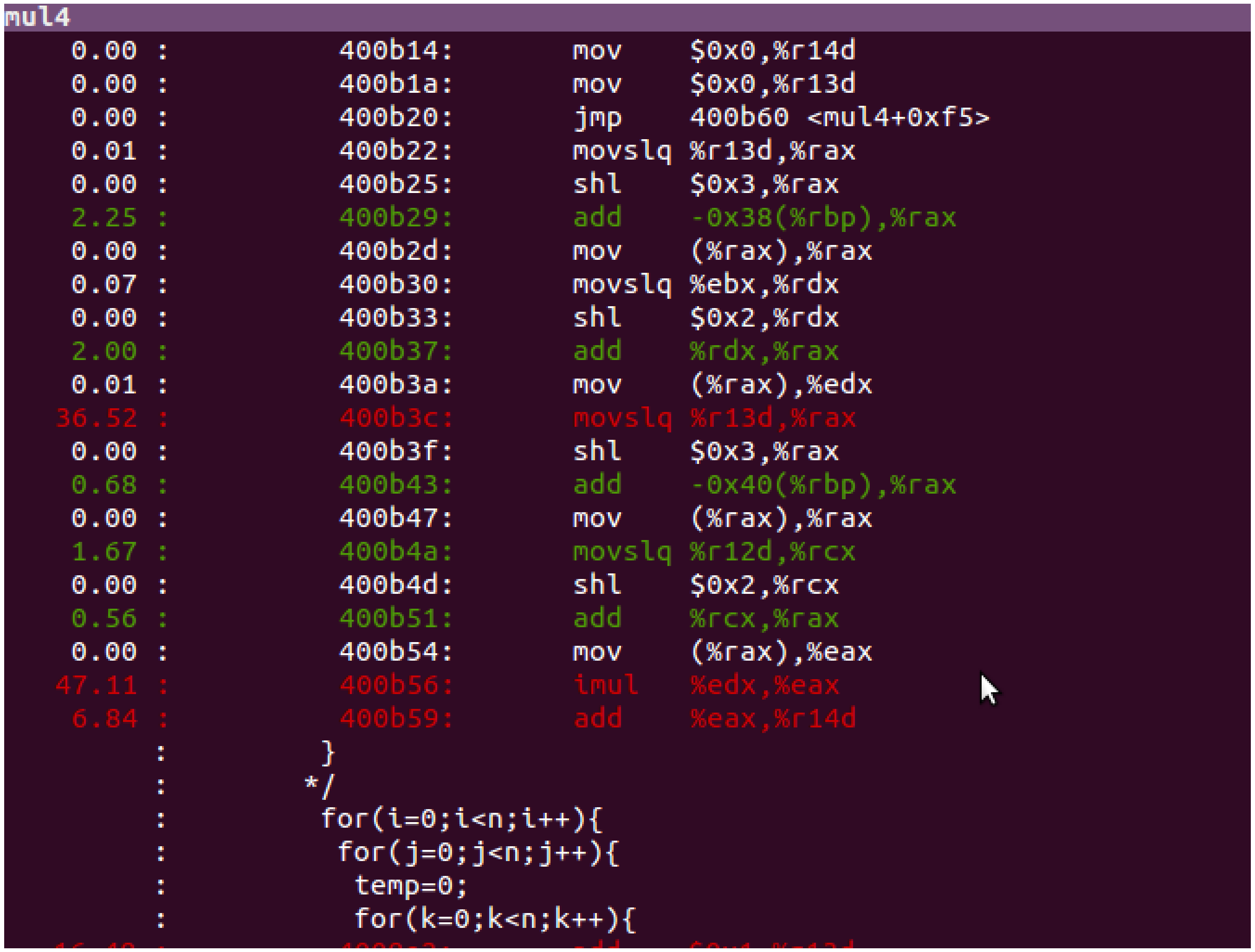
```
adsingh1@ubuntu:~/1200884286H1/Matrix Multiplication$ perf stat -e cache-misses ./matrixmul 1000 3

Time taken by mul3 for array size 1000 is 2095 microseconds

 Performance counter stats for './matrixmul 1000 3':

         1,943,723 cache-misses

       2.114881840 seconds time elapsed

adsingh1@ubuntu:~/1200884286H1/Matrix Multiplication$ perf stat -e cache-misses ./matrixmul 1000 4

Time taken by mul4 for array size 1000 is 15046 microseconds

 Performance counter stats for './matrixmul 1000 4':

        60,013,331 cache-misses

      15.064431169 seconds time elapsed
```

Event Based Sampling Example



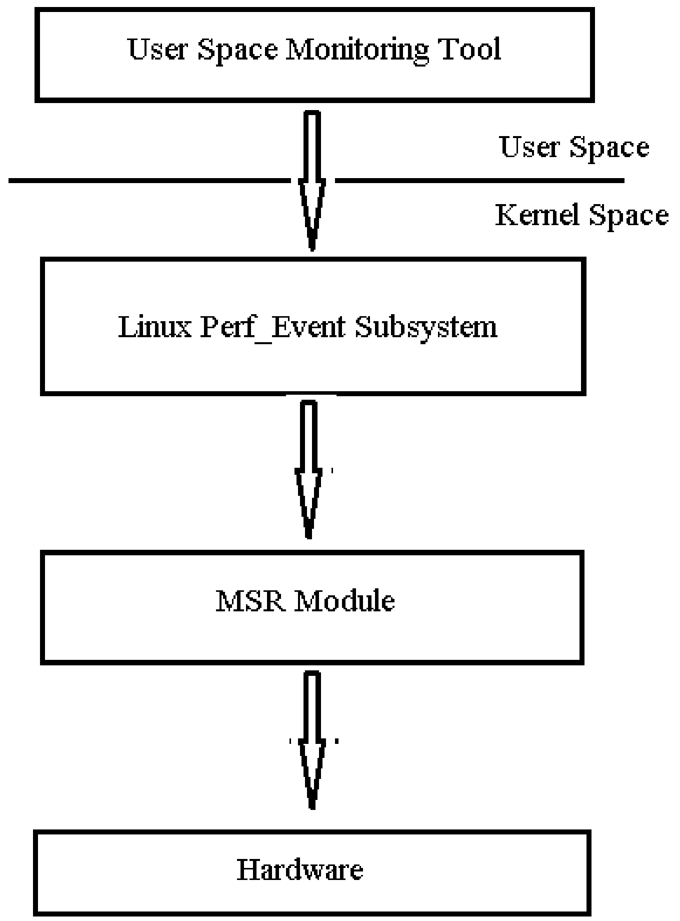
```
Events: 15K cycles
 99.62%  matrixmul  matrixmul          [.] mul4
  0.06%  matrixmul  matrixmul          [.] transposeMatrix
  0.04%  matrixmul  matrixmul          [.] main
  0.02%  matrixmul  [kernel.kallsyms]  [k] periodic_unlink
  0.02%  matrixmul  [kernel.kallsyms]  [k] scan_periodic
  0.01%  matrixmul  [kernel.kallsyms]  [k] update_rq_clock
  0.01%  matrixmul  [kernel.kallsyms]  [k] ioread32
  0.01%  matrixmul  [kernel.kallsyms]  [k] account_user_time
  0.01%  matrixmul  [kernel.kallsyms]  [k] ehci_irq
  0.01%  matrixmul  [kernel.kallsyms]  [k] __do_softirq
  0.01%  matrixmul  [kernel.kallsyms]  [k] check_preempt_wakeup
  0.01%  matrixmul  [kernel.kallsyms]  [k] tick_do_update_jiffies64
  0.01%  matrixmul  [kernel.kallsyms]  [k] internal_add_timer
  0.01%  matrixmul  [kernel.kallsyms]  [k] irq_entries_start
```

Event Based Sampling Example

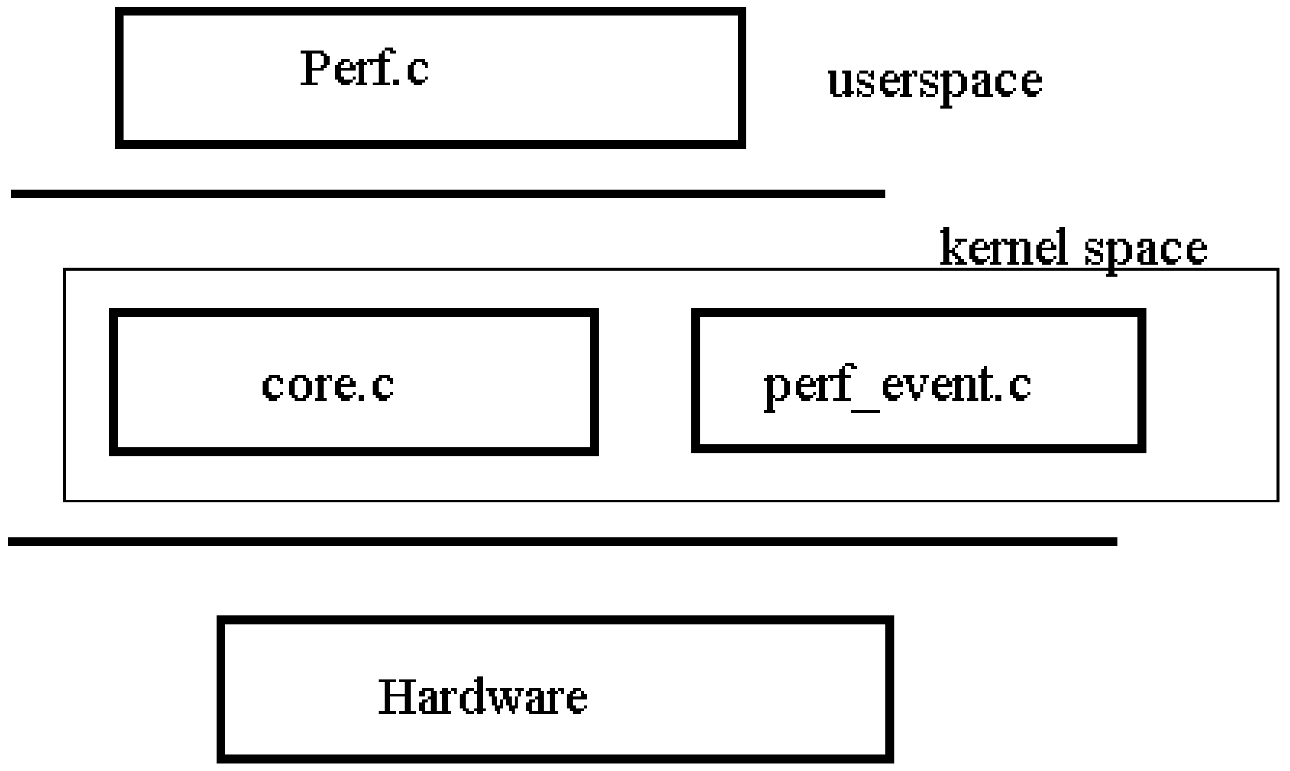


```
mul4
    0.00 :          400b14:       mov    $0x0,%r14d
    0.00 :          400b1a:       mov    $0x0,%r13d
    0.00 :          400b20:       jmp    400b60 <mul4+0xf5>
    0.01 :          400b22:       movslq %r13d,%rax
    0.00 :          400b25:       shl    $0x3,%rax
    2.25 :          400b29:       add    -0x38(%rbp),%rax
    0.00 :          400b2d:       mov    (%rax),%rax
    0.07 :          400b30:       movslq %ebx,%rdx
    0.00 :          400b33:       shl    $0x2,%rdx
    2.00 :          400b37:       add    %rdx,%rax
    0.01 :          400b3a:       mov    (%rax),%edx
   36.52 :          400b3c:       movslq %r13d,%rax
    0.00 :          400b3f:       shl    $0x3,%rax
    0.68 :          400b43:       add    -0x40(%rbp),%rax
    0.00 :          400b47:       mov    (%rax),%rax
    1.67 :          400b4a:       movslq %r12d,%rcx
    0.00 :          400b4d:       shl    $0x2,%rcx
    0.56 :          400b51:       add    %rcx,%rax
    0.00 :          400b54:       mov    (%rax),%eax
   47.11 :          400b56:       imul   %edx,%eax
    6.84 :          400b59:       add    %eax,%r14d
         :          }
         :         */
         :          for(i=0;i<n;i++){
         :           for(j=0;j<n;j++){
         :            temp=0;
         :            for(k=0;k<n;k++){
```

Architecture of Linux ‘Perf’



Linux Perf_event Subsystem



Linux Kernel Support for Perf

Some important Data Structures are

- Struct task_struct {

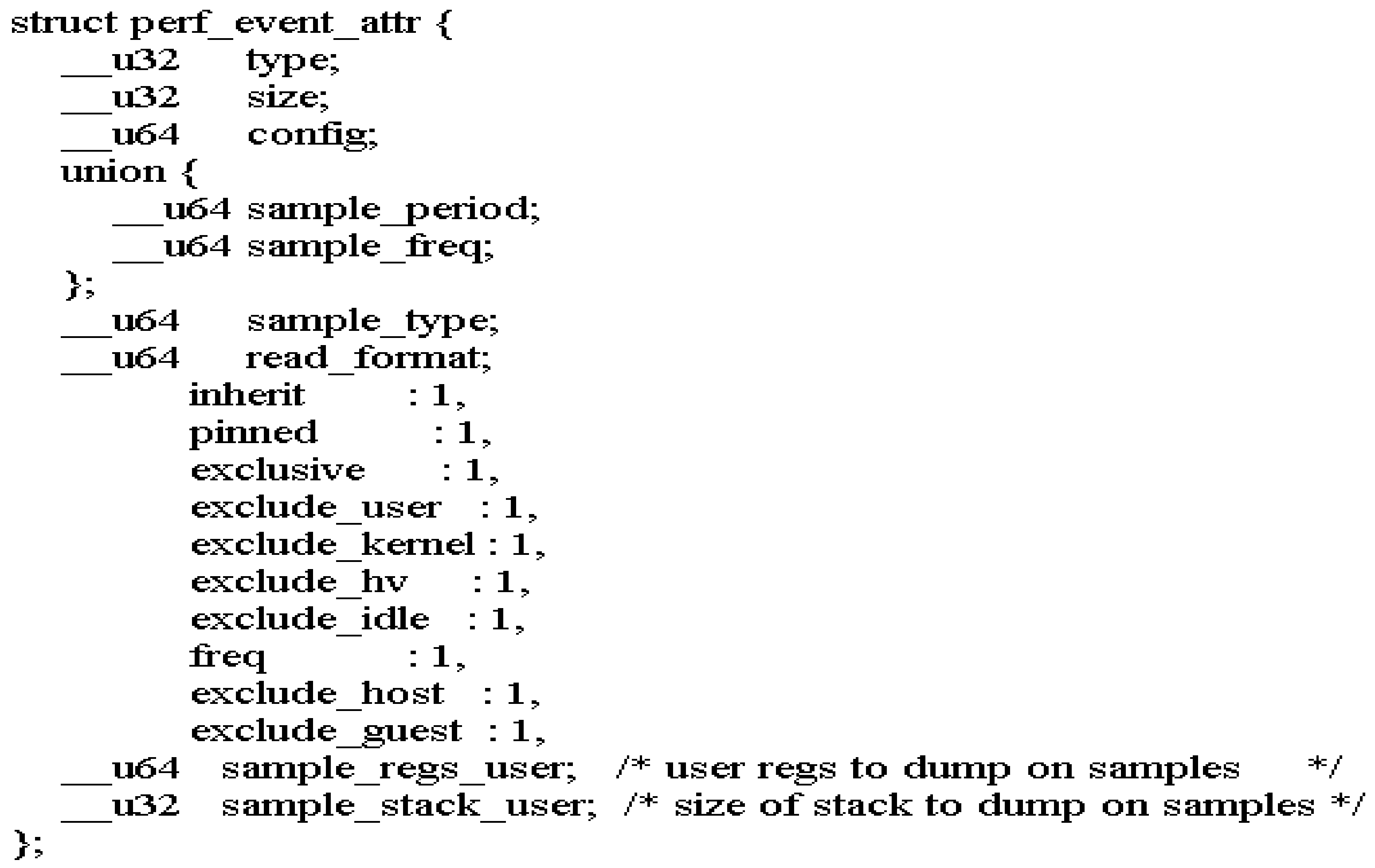
 struct perf_event_context
   *perf_event_ctxp[perf_nr_task_contexts];

….

}

Struct perf_event_attr ;

Important Fields in Struct perf_event_attr.



```
struct perf_event_attr {
    __u32     type;
    __u32     size;
    __u64     config;
    union {
        __u64 sample_period;
        __u64 sample_freq;
    };
    __u64     sample_type;
    __u64     read_format;
              inherit       : 1,
              pinned        : 1,
              exclusive     : 1,
              exclude_user  : 1,
              exclude_kernel : 1,
              exclude_hv    : 1,
              exclude_idle  : 1,
              freq          : 1,
              exclude_host  : 1,
              exclude_guest : 1,
    __u64   sample_regs_user;   /* user regs to dump on samples     */
    __u32   sample_stack_user;  /* size of stack to dump on samples */
};
```

Perf

- The Perf gives a file descriptor for every event that we want to monitor.

  int perf_event_open(struct perf_event_attr ****attr*, pid_t** *pid***, int** *cpu***, int** *group_fd***, unsigned long** *flags***);**

- Using this file descriptor we perform read and write on the events.
- Events can be enabled and disables using ioctl system calls.

- In case of sampling when the interrupt is generated, the interrupt is processed by an NMI handler when calls the *perf_event_overflow()* routine which collects the samples as defined in the perf_event_attr register.

Process Monitoring

- Perf can count per-thread or per-process event counts. This is achieved by loading and restoring the events counts during context.
- When the context switch we call *prepare_task_switch()* function which stores the values in the task_struct for that process.
- Once the context switch is over we call finish_task_switch() function which loads the values of the events from the task_struct on the hardware.

Limitations

- Too Few Counters
- Speculative Counts
- Sampling Delay
- Multiplexing of events

Q&A

Thanks