A Study of Performance Monitoring Unit, *perf* and *perf_events* subsystem

Team
Aman Singh
Anup Buchke

Mentor
Dr. Yann-Hang Lee

Summary

Performance Monitoring Unit, or the PMU, is found in all high end processors these days. The PMU is basically hardware built inside a processor to measure it's performance parameters. We can measure parameters like instruction cycles, cache hits, cache misses, branch misses and many others depending on the support i.e. hardware provide by the processor. And as the measurement is done by the hardware there is very limited overhead.

In this review we will look inside the PMU of Intel IA-32 and see how it works, from a very high level. How these PMU registers are configured and how they can be utilized for different types of performance measurements. We will then look into how the software utilizes this hardware. We will closely look into the working of the linux command line utility: *perf* and the *perf_events* subsystem.

Contents

PMU Hardware

The PMU in the Intel IA-32 architecture consists of two types of registers or Model-Specific Registers (MSRs). They are called model specific because every processor model has some registers different from the other models, even from the same company. These two types are the Performance Event Select Registers and the Performance Monitoring Counters(PMC). Measuring a performance event requires programming the Event Select Registers. The performance events are counted in the PMCs. So for measuring a performance event we need both the event selector and the PMC.

Layout of IA32_PERFEVTSELx MSRs:

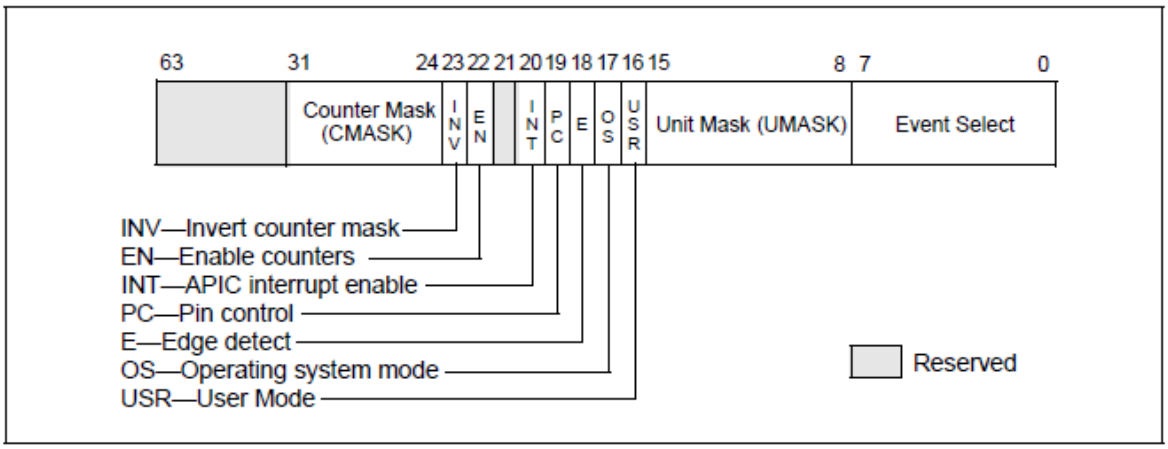

Figure 1: Layout of IA32_PERFEVTSELx MSRs from the Intel64_developer_model_vol3b.pdf

- **Event Select Fields (bits 0-7) :**
  This field is used to select the logic unit to detect the performance monitoring event to be monitored. So the values to be filled in this field is determined by the architecture.

- **Unit mask (UMASK) fields (bits 8-15):**
  The logic unit selected by the Event Select field might be capable of monitoring multiple events. So this UMASK field is used to select one of those events which can be monitored by the logic unit. So based on the logic unit selected the UMASK field may have one fixed value or multiple values, which is dependent on the architecture.

- **USR flag (bit 16):**
  This flag, if set, tells the logic unit to monitor events which happen when the processor is running in the User privilege level i.e. levels 1 through 3.

- **OS Flag (bit 17):**
  This flag, if set, tells the logic unit to monitor events which happen when the processor is running in the highest privilege level i.e. level 0. This flag and the USR flag can be used together to monitor or count all the events.

- **E (Edge Detect) (bit 18):**
  This flag when set counts the number of times the selected event has gone from low to high state.

- **PC (Pin Control) (bit 19):**
  This flag when set increments the counter and toggles the PMi pin when the monitored event occurs. And when not set, it toggles the PMi pin only when the counter overflows.

- **INT (APIC interrupt enable) flag (bit 20):**
  When this flag is set the processor raises an interrupt when the performance monitoring counter overflows.

- **EN (Enable Counters) flag (bit 22):**
  This flag when set enables the performance monitoring counters for the event and when clear disables the counters.

- **INV (inversion) flag (bit 23):**
  This flag when set inverts the output of CMASK comparison. This enables the user to set both greater than and less than comparisons between CMASK and the counter value.

- **CMASK (Counter mask) field (bits 24 through 31):**
  If this field has a value more than zero then that value is compared to the number of events generated in one clock cycle. If the events generated is more than the CMASK value then the counter is incremented by one else the counter is not incremented.

Other features provided by hardware:

- **Fixed function performance counter register and associated control register:**
  There are a few counters which can measure only a specific event unlike general purpose counters which can be configured to measure different events.

- **Global Control Registers:**
  Some architectures provide global control registers which can be used to control all or a group of control registers or counters. This reduces the number of instructions required to modify the control registers and hence eases programming.

Types of Performance Measurement

Counting

In this type of measurement the total number of events that happen in a given time duration are aggregated and reported at the end of the duration. The performance control registers are set for counting the desired event and after the end of monitoring period the values of these registers are read.

Challenges faced:

- If the number of events to be monitored are more than the total number of counters provided by the processor.
- If two different events to be monitored are measured by the same digital logic present in the processor.

Resolution:

Multiplexing is used to resolve the issues mentioned above.

- In the first case, as the number of counters is less the events share time of the same counter i.e. time division multiplexing. Which means that one event does not get a dedicated counter for the entire duration of measurement. Instead the events are measured in small durations many times during the entire measuring period. At the end of measurement duration the actual measurement period is also recorded and the aggregated event count is scaled for the complete measurement period.
- In the second case, the same technique is used as in the first case. The only difference being that this time the digital logic unit is time multiplexed to measure different events.

Limitations:

- Although multiplexing solves some issues and will be pretty close to the actual values but the scaled results are not completely reliable. It may so happen that the event which was not being measured for a particular instance may have spiked or tanked during that instance and the scaled value will be misleading.
- So, in cases where highly precise values are desired the user should take care that the monitored event gets dedicated hardware and is not time multiplexed.

Example:

Here is an example of the counting type of measurement. Here we are measuring the number of cache misses for two different types of matrix multiplications. We have designed the first algorithm to be more efficient than the second one by utilizing spatial locality for memory references hence the cache hits for the first one is expected to be much lower than the second algorithm. We have measured the total number of cache hits for both the multiplication algorithms using the *perf* tool with *stat* option.

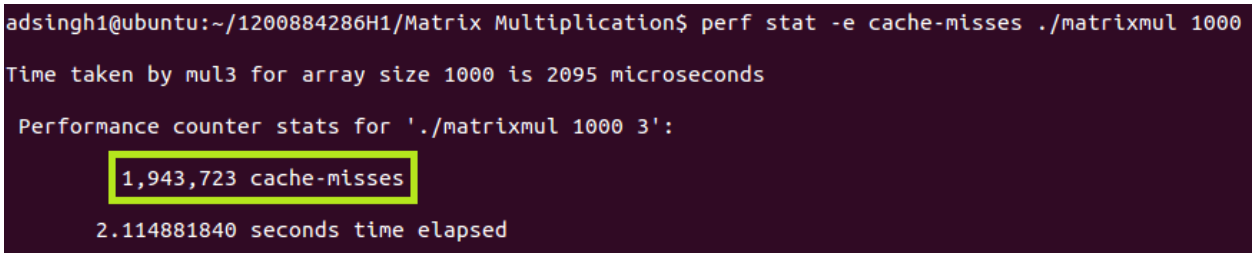

Figure 2a: Performance Measurement using Counting Example 1 with efficient matrix multiplication algorithm

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

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
```

Figure 2b: Performance Measurement using Counting Example 2 with inefficient matrix multiplication algorithm

Event based Sampling

In this type of measurement, the PMU counters are configured to overflow after a preset number of events and when the overflow happens the process status information is recorded by capturing the data of the instruction pointer, general purpose registers and EFLAG registers. This sampled data can be utilized for profiling software applications, finding how the software is utilizing the underlying hardware and many other purposes.

Limitations:

- **Sampling Delay**
  There is delay between the counter overflow and the time when interrupt is raised. This combined with the long pipelines present in high-end processors the program counter data stored at the time of sampling may not be the event which caused the counter to overflow.
- **Speculative count**
  Most high end processors these days use branch predictions which leads to speculative execution of instructions which may not complete if some other branch is selected. But these speculatively executed instructions may cause events and contribute to the event count even if they do not complete, which is not correct.

Example:
In the example, discussed in the previous section, we saw that the second algorithm is not performing well. So let us now use the *perf record* utility to find out approximately where a lot of execution time is being spent in that program. Below is the snapshots of the output of *perf record* interpreted using *perf report*.

```
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
```

Figure 3a: Example of performance monitoring using sampling

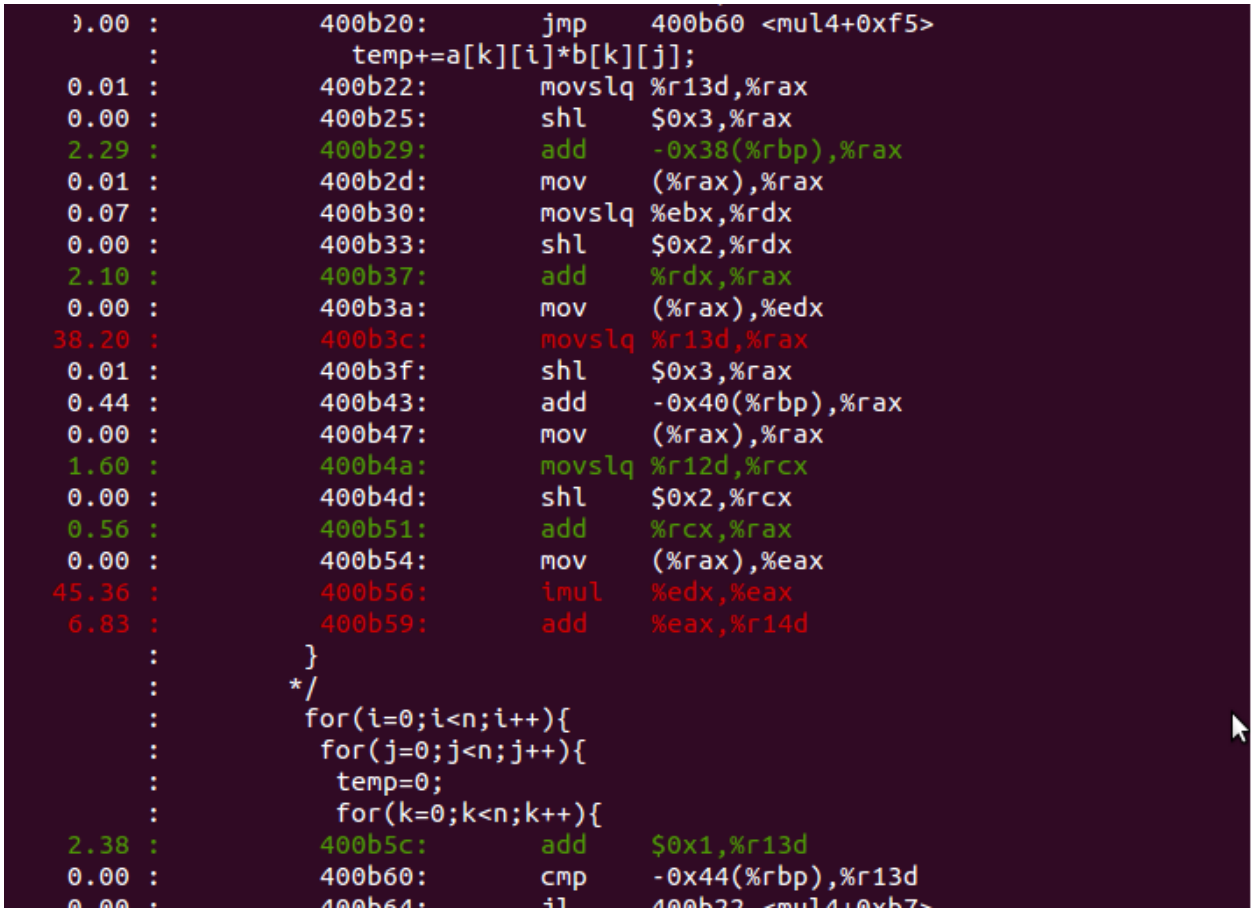

Figure 3b: Example of performance monitoring using sampling

We can see from figure 3a that 99.62% of samples, which were taken using the default event cycles, are in the mul4 function which is the function which multiplies the matrix. On further digging we can see from Figure 3b that a lot of time is spent is moving the data which is because of the many cache hits. So using this technique performance issues in software applications can be debugged easily.

Performance Monitoring Tools

This following is the list of performance monitoring tools which are widely used.

- Intel VTUNE Amplifier
- OProfile
- LTTng6
- Perf Tools

perf - Command Line Utility

This is an open source tool available in linux which can be used for monitoring performance of applications.

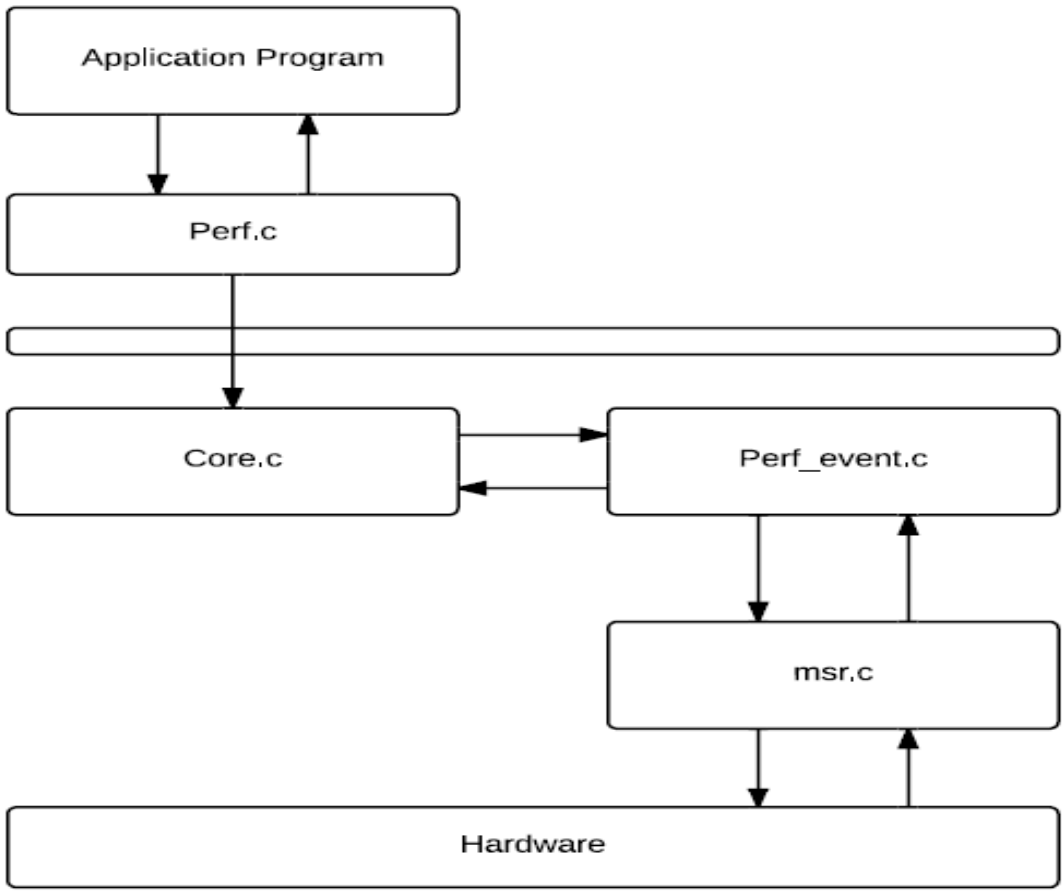

Figure 4: Architecture of perf event subsystem

The Linux Perf_Event Subsystem consists of the files core.c and perf_event.c. These files are the interface between the linux kernel and various user space performance monitoring tool.

Data Structures in Linux perf_event Subsystem

The following are some of the important data structures which are used by the perf_event subsystem.

```
struct perf_event;
struct perf_event_attr;
struct perf_event_context;
struct perf_sample_data;
struct pmu;
```

Important Fields in the Data Structures

```
struct perf_event {
struct perf_event          *group_leader;
struct pmu                 *pmu;
u64                        total_time_enabled;
u64                        total_time_running;
struct perf_event_attr     attr;
atomic64_t                 child_count;
struct perf_event_context  *ctx;
perf_overflow_handler_t    overflow_handler;
struct task_struct         *owner;  }
```

Description:

group_leader

This field specifies the leader of the group of events attached to the process.

pmu

This field points to the generic performance monitoring unit structure.

total_time_enabled

This field specify the total time in nanoseconds that the event has been enabled.

total_time_running

This field specify total time in nanoseconds that the event is running(scheduled onto the CPU)

owner

This field points to the task structure of the process which has monitoring this event.

```
struct perf_event_attr {
__u32     type;
__u64     config;
__u64 sample_period;
__u64 sample_freq;
__u64 sample_type;
exclusive      : 1,
exclude_user   : 1,
exclude_kernel : 1,
exclude_hv     : 1,
exclude_idle   : 1,
exclude_host   : 1,
exclude_guest  : 1  }
```

Description:

type

This field specifies the overall event type.

config

This field specifies which event needs to be monitored. It is used along with type to decide the exact event.

sample_period, *sample_freq*

Sampling period defines the N value where N is the number of events after which the interrupt is generated. It can be counted in terms of frequency as well.

sample_type

The various bits in this field specify which values to include in the sample.

exclude_user

This bit when enabled the count excludes the user-space events.

exclude_kernel

This bit when enabled the count exclude the kernel-space events.

```
struct perf_event_context {
struct list_head              event_list;
int                           nr_events;
struct perf_event_context     *parent_ctx;
u64                           time;
u64                           timestamp; }
```

Description:

event_lists

This field specifies the list of events.

nr_events

This field specifies the number of events that are currently monitored.

parent_ctx

This fields points to the context of the processes parent.

time,timestamp

These are context clocks, they run when the context is enabled.

```
struct pmu {
void (*pmu_enable)            (struct pmu *pmu);
void (*pmu_disable)           (struct pmu *pmu);
void (*start)                 (struct perf_event *event, int flags);
void (*stop)                  (struct perf_event *event, int flags);
void (*read)                  (struct perf_event *event); }
```

Description:

This structure majorly contains the function pointers to various PMU related functions.

pmu_enable,pmu_disable

These functions are used to fully disable/enable a PMU.

start,stop

These functions are used to start or stop a counter on a PMU.

read

This function is used to update the event value for a particular counter.

Counting Support in Linux Perf

perf_event assigns one file descriptor per event and either per-thread or per-CPU. The system call *perf_event_open()* configures the hardware MSRs and creates a file descriptor which can be used for reading the performance measurement data. Once the file descriptor is obtained we can issue subsequent read calls to get the values of the performance counters. These values are then aggregated at the end of the program execution.

The following is the execution flow for getting the file descriptor.

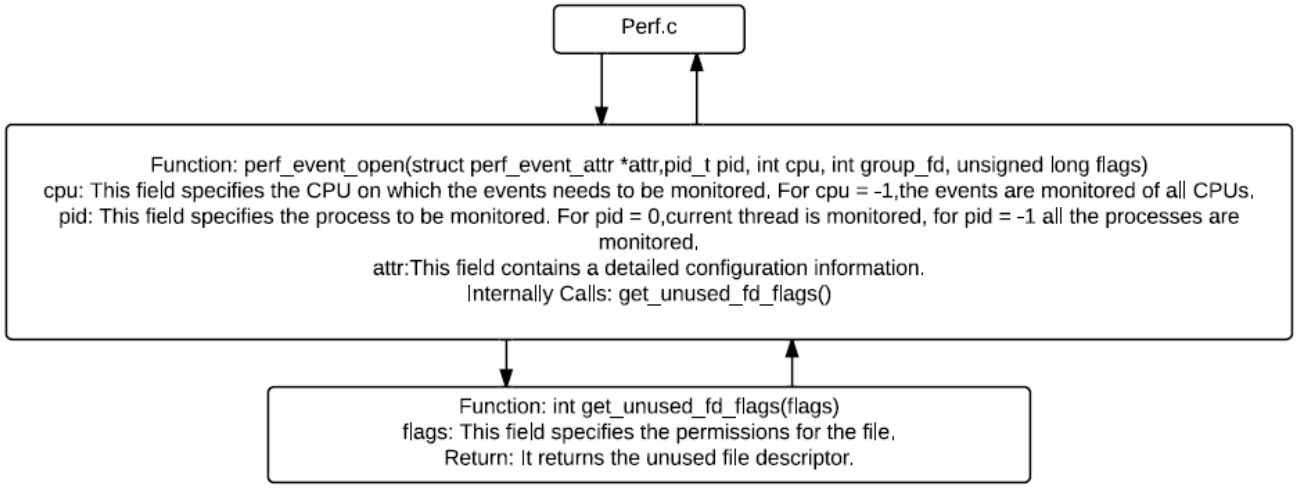

Figure 5: function call trace

For enabling and disabling performance monitoring events we use the ioclt and prctl system calls.

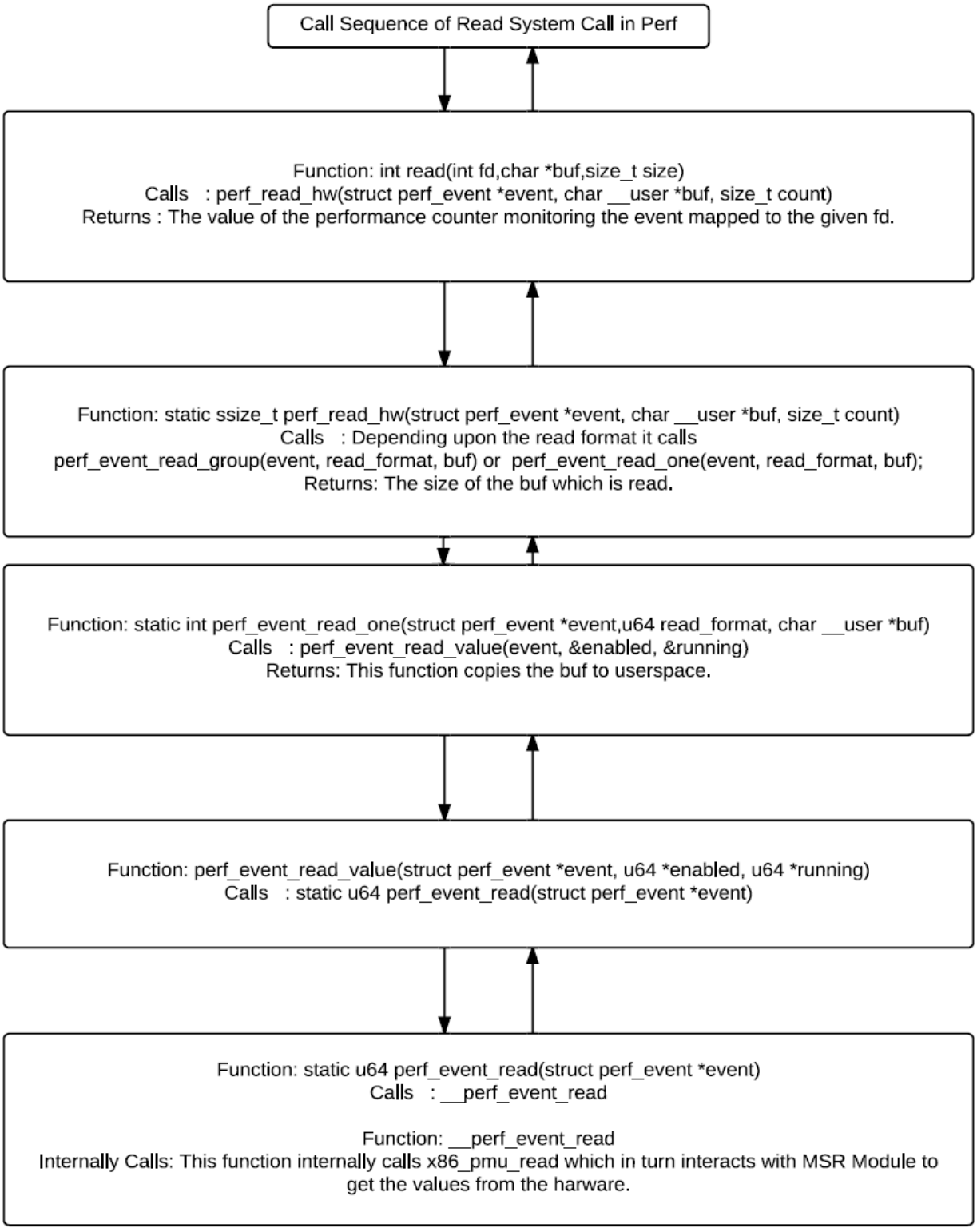

Figure 6: Execution flow of the read system call

Sampling Support in Perf

Perf_event is a Linux Subsystem. When the linux kernel is loaded the corresponding perf modules are statically loaded.When the perf subsystem is loaded it register a Nonmaskable interrupt handler to process its events.This interrupt is raised when the counter overflows.The interrupt handler takes a snapshot of the registers and the counters are reset to predefined values.

In Linux the following functions are being invoked:

1. perf_event_nmi_handler()
2. This invokes x86_pmu_handle_irq().
3. This function performs the following steps
   3.1. Iterate through all the performance events.
   3.2. Extract the values and check if the counters overflow.(Uses perf_event_overflow() function for it)
   3.3. if the counter has overflow stop the counting using the function x86_pmu_stop().

Measuring Software Events and Per-process performance using perf.

For measuring per-thread or per-process performance, the perf_event subsystem invokes some functions of the Linux scheduler. For every context switch the context of current events are been pushed on the task_struct structure. Once the context switch is over the events attached to the newly scheduled process are accessed via the *current* macro in Linux which point to the currently running process.

In the Linux context switch the following activities happens.

1) The function static inline void context_switch(struct rq *rq, struct task_struct *prev, struct task_struct *next) is invoked by the scheduler.

2) prepare_task_switch(struct rq *rq, struct task_struct *prev,struct task_struct *next) is invoked.

3) This function invokes perf_event_task_sched_out(prev, next)

4) This function performs two actions. It counts the software event of context switch and also calls __perf_event_task_sched_out which copies the context into the task_struct for the task which is scheduled out.

5) Then switch_to performs the task swapping.

6) Once the context switch is done the function finish_task_switch() is invoked. This function invokes perf_event_task_sched_in() which using the current macro in Linux access the newly scheduled task's task_struct and load the monitoring events on the CPU.

References

1. http://www.eece.maine.edu/~vweaver/projects/perf_events/perf_event_open.html
2. http://en.wikipedia.org/wiki/List_of_performance_analysis_tools
3. http://lxr.free-electrons.com
4. A Performance Counter Architecture for Computing Accurate CPI Components.
   Stijn Eyerman Lieven Eeckhout ELIS, Ghent University, Belgium
   {seyerman,leeckhou}@elis.UGent.be, Tejas Karkhanis James E. Smith
   ECE, University of Wisconsin–Madison {karkhani,jes}@ece.wisc.edu
5. Are Hardware Performance Counters a Cost Effective Way for Integrity Checking of Programs?
   Corey Malone , Mohamed Zahran, Ramesh Karri
6. https://perf.wiki.kernel.org/index.php/Main_Page
7. http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html