KERNEL PROGRAMMING

Project Report

Performance monitoring using PMU

Author:
Anubhav Sharma

Supervisor:
Prof. Puroshottam Kulkarni

April 29, 2016

Abstract

Performance monitoring counters can be used to analyze and measure performance of a program in a system. Most modern processors contain such counters. These counters are nothing but hardware registers which measure programmable events in the processor. The aim of this project is to use these hardware counter registers to obtain various statistics about a program such as - relative distribution of number of branch misses or L1 cache misses or CPU cycles in every function. The approach used to gather statistics is sampling based, in the sense that we periodically take samples of the state of the program to generate the statistics. The tool can also be used to adopt a counting based approach in gathering statistics as we shall see.

1 Project Approach

The Intel(R) Core(TM) processor provides two types of counters - fixed counters and programmable counters. Programmable counters are general purpose counters which can be programmed to measure different types of events. These counters are incremented whenever a specified event takes place. These performance monitoring counters are supported by performance event select registers. These registers specify what events to measure, at which privelege level, and whether to generate interrupts or not.

In this project we have used the counter register `PERF_PMC0 (0x00C1)` to keep track of the count of events and `PERF_EVENTSEL0 (0x186)` to select the event to count. These registers are model specific i.e. every processor model could have a different name or address for them.

Figure 1 shows the layout of the `PERF_EVENTSEL0` register.

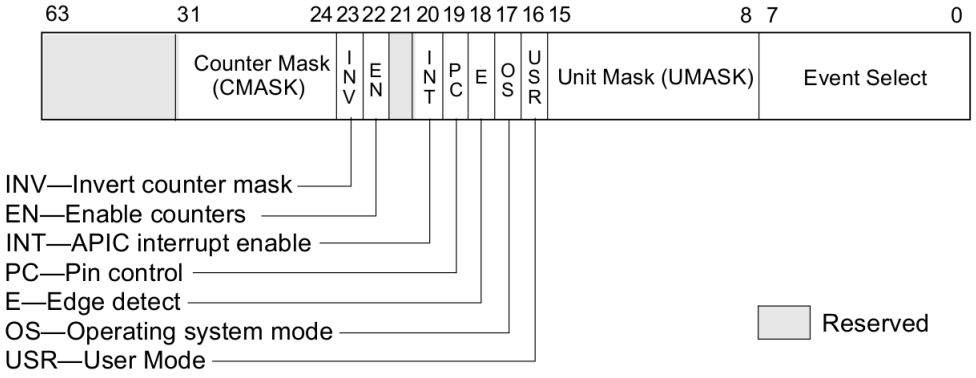

Figure 1: Layout of PERFEVTSELx MSRs

The event select field combined with the unit mask can be used to select which events to measure. For example, to count the number of mispredicted branches, we need to set event select as `0xc5` and unit mask as `0x00`. The USR flag 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. The OS 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. The INT flag, when set, the processor raises an interrupt when the performance monitoring counter overflows. The EN flag when set enables the performance monitoring counters for the event and when clear disables the counters.

The first step is to set up the event select register appropriately. The next step is to setup an interrupt handler which gathers samples of the state of a program whenever a counter overflow event occurs. This is done using the register_nmi_handler function. In the interrupt handler we store the state of the program in execution which would be passed to user space for analyzing. Finally, when the program we are profiling exits, we allow any user space program to collect the samples and analyze them. This is done by using a character device.

2 Project Design

The whole project can be broadly divided into three components - a user space program which contains the code to be profiled, a kernel module which contains the logic to setup counters and handle interrupts, and a shell script which gathers all the samples and generates a statistics out of them. Figure 2 shows the whole workflow of the project on a very high level.

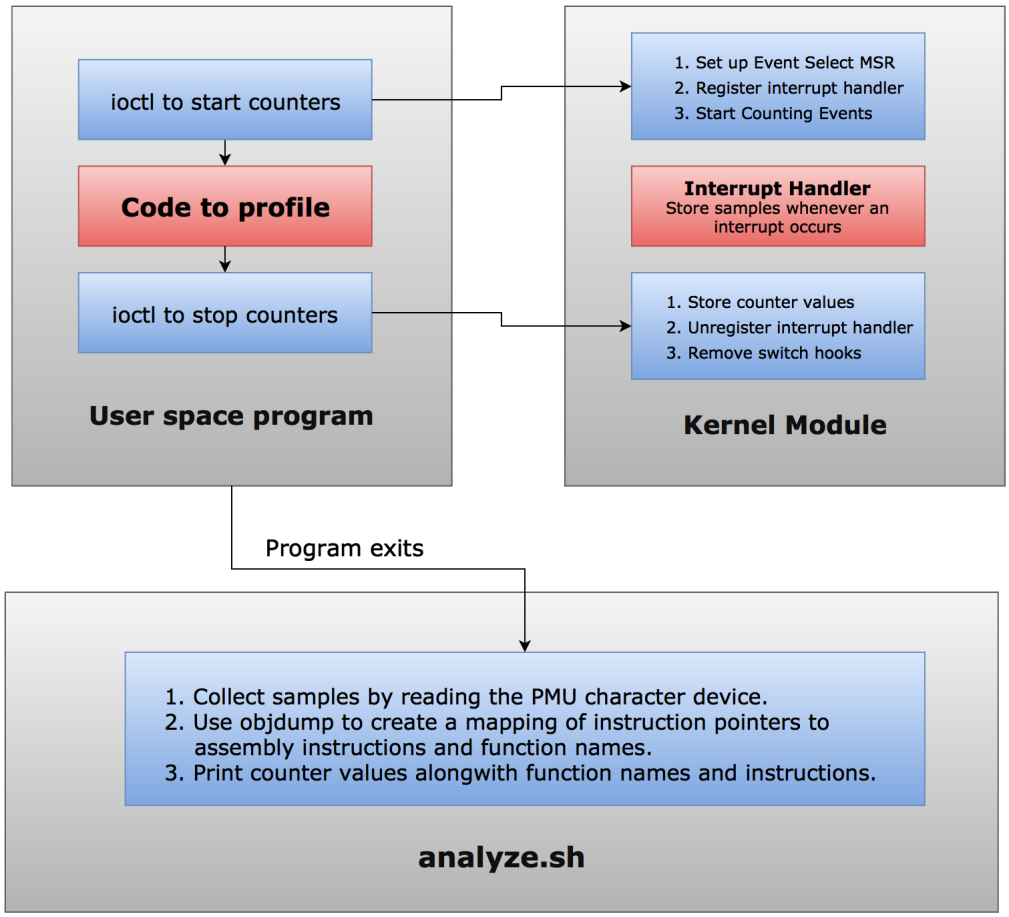

Figure 2: Project Workflow

The kernel module provides an ioctl interface to allow user programs to interact with kernel space. The ioctl interface supports two commands - `PMU_START_COUNTERS` and `PMU_STOP_COUNTERS`. A user program can use the `PMU_START_COUNTERS` command in ioctl to setup the hardware registers for starting code profiling. The call should provide information such as sampling

rate and code of the event to be measured. The kernel module upon receiving such a call, would setup the event select and counter registers appropriately. It would also register an interrupt handler which would be called whenever the counter overflows. Since we need to only count events for the process that started the counters, we need to make sure that the PMU does not count these events for other processes. To achieve this, we have added two hooks in the kernel - one for intercepting context switches and other for intercepting task exits. Whenever our process is switched out of the CPU, we disable the counters. The counters are enabled when our process is run again. This ensures that the processor is only counting events for the process that made the ioctl call. The counters and interrupt handler is disabled when our process exits.

We maintain an array of instruction pointers throughout the sampling period. New entries are added into this array whenever the interrupt handler is called. The rate at which the interrupt handler will be invoked is controlled by a variable `samplingRate`. Since an interrupt is triggered when a counter overflows, we can control the frequency of such overflows by initializing the counter with some large values. For example, if we put `(MAX_COUNTER_VALUE-1000)` into a counter register, it would overflow after `1000` events have occurred. At this point we again put `(MAX_COUNTER_VALUE-1000)` into the register to make sure that the next interrupt also occurs after `1000` events.

When the user program make an ioctl call to stop the counters using the `PMU_STOP_COUNTERS` command, the kernel module disables the counters and unregisters the interrupt handler. The user can then use the read function call on the character device to get a list of samples collected during the profiling. These samples are used by the shell script analyze.sh which generates a distribution of event occurrences per instruction.

3 Implementation Details

In this section, we would look at the implementation of the kernel module. Note that the code shown here is heavily trimmed and oversimplified to make it more readable and understandable.

init_module()

```
struct file_operations pmufops = {
  .unlocked_ioctl = pmu_ioctl,
  .read = device_read
};
int perfModuleInit(void) {
  int major = register_chrdev(261, "pmuDriver", &pmufops);
  if (major < 0) {
    return -1;
  }
  return 0;
}
```

The`init_module()` function registers a character device `pmuDriver` and overrides it read and ioctl function. This device driver is used by user space programs to interact with the kernel.

pmu_ioctl()

```
struct pmuStruct {
  int eventToSample;
  int pid;
  int samplingRate;
};

static long pmu_ioctl(struct file *f, unsigned int cmd,
   unsigned long arg) {
  if (cmd == PMU_START_COUNTERS) {
    struct pmuStruct pst = *((struct pmuStruct *) arg);
    return requestEventCounter(pst);
  } else if (cmd == PMU_STOP_COUNTERS) {
    disableCounters(cpu);
    u64 counterValue = readCounterValue();
    *((unsigned long long *) arg) = counterValue;
  }
  return 0;
}
```

The ioctl interface provides two options - start counters and stop counters. A user's request for starting counters is sent to the function `requestEventCounter` alongwith the `pmuStruct` structure which contains information such as the event to sample and the rate of sampling. If a user requests for stopping the counters, the counters are disabled and the user is returned the current counter value.

requestEventCounter()

```
int requestEventCounter(struct pmuStruct pst) {
  myTask = pid_task(find_vpid(pst.pid), PIDTYPE_PID);
  u64 eventToCount = pst.eventToSample;
  sampleInterval = (u64) pst.samplingRate;

  setupCounters(eventToCount, sampleInterval);
  instructionPointerList = vmalloc(LIST_SIZE);
  registerOverflowHandler(sampleInterval);

  contextSwitchHook = contextSwitchHookFunction;
  taskExitHook = taskExitHookFunction;
  return 0;
}
```

The `requestEventCounter()` function first finds the task_struct of the process to be profiled. The counters are then setup with appropriate event stored in `eventToCount` and sampling rate stored in `sampleInterval`. The `instructionPointerList` variable is an array which stores the list of instruction pointer values whenever a counter overflow interrupt occurs. The function then registers an interrupt handler which would be called whenever a counter overflows. We also initialize the context switch and task exit hook functions here. The `contextSwitchHookFunction()` is called whenever a process is switched in or out. The `taskExitHookFunction` is called whenever any process terminates.

overflowHandler()

```
static int overflowHandler(unsigned int cmd, struct pt_regs
    *regs) {
  instructionPointerList[++index] = task_pt_regs(myTask)->
     ip;
  wrmsrl_safe_on_cpu(cpu, MSR_ARCH_PERFMON_PMC0, (u64)
     sampleInterval);
}
```

The `overflowHandler()` function stores the instruction pointer of our task in the array of instruction pointers. It then writes the initial value of `sampleInterval` back to the `PMC0` register counter. This ensures that the register would overflow when `sampleInterval` number of events have occurred.

contextSwitchHookFunction()

```
void contextSwitchHookFunction(struct task_struct *prev,
    struct task_struct *next) {
  if (prev->pid == myTask->pid) {
    disableCounters(cpu);
  }
  if (next->pid == myTask->pid) {
    enableCounters(cpu);
  }
}
```

The `contextSwitchHookFunction()` simply disables the counters whenever our process is scheduled out of the CPU and enables the counters whenever our process is scheduled in. The hook was added in the `_schedule()` function defined in `linux/kernel/sched/core.c` just before the `contex_switch` call is made. The `taskExitHook` was added in the `do_exit()` function in `linux/kernel/exit.c`.

Apart from the functions described there are some helper functions that allow to read/write/clear/enable/disable counters that make use of the `rdmsrl` and `wrmsrl` instructions for reading and writing into model specific registers.

4 Experiments

We compare the statistics given by our tool with perf to validate the correctness of our results.

4.1 Test 1

The following user program was profiled for number of branch mispredictions:

test1.cpp

```
static unsigned long testFunction(unsigned long n) {
  const unsigned arraySize = 32768;
  int data[arraySize];

  for (unsigned c = 0; c < arraySize; ++c)
    data[c] = std::rand() % 256;

  unsigned long sum = 0;
  for (unsigned i = 0; i < 100000; ++i) {
    for (unsigned c = 0; c < arraySize; ++c) {
      if (data[c] >= 128)
        sum += data[c];
    }
  }
  return sum;
}
```

The results given by perf are as follows:

```
 1.14 |7d:   mov    -0x20010(%rbp),%eax
 0.00 |       mov    -0x20000(%rbp,%rax,4),%eax
 0.16 |       cmp    $0x7f,%eax
      |     ↓ jle    a5
 0.01 |       mov    -0x20010(%rbp),%eax
 0.16 |       mov    -0x20000(%rbp,%rax,4),%eax
49.39 |       cltq
 1.36 |       add    %rax,-0x20008(%rbp)
 0.41 |a5:    addl   $0x1,-0x20010(%rbp)
 8.65 |ac:    cmpl   $0x7fff,-0x20010(%rbp)
38.72 |     ↑ jbe    7d
```

Figure 3: Perf Output

The results given by our module are as follows. The first column indicates the relative distribution of events that occurred in that particular instruction. Its similar to the first column of the perf output. The second column shows the name of the function and the last columns show the actual assembly instructions.

```
48.0     _ZL12testFunctionm      4010ad  cltq
37.1     _ZL12testFunctionm      4010c7  jbe 40108e <_ZL12testFunctionm+0x7d>
10.2     _ZL12testFunctionm      4010bd  cmpl $0x7fff,-0x20010(%rbp)
1.8      _ZL12testFunctionm      4010af  add %rax,-0x20008(%rbp)
1.6      _ZL12testFunctionm      40108e  mov -0x20010(%rbp),%eax
0.9      _ZL12testFunctionm      4010b6  addl $0x1,-0x20010(%rbp)
0.3      _ZL12testFunctionm      40109b  cmp $0x7f,%eax
0.1      _ZL12testFunctionm      4010a6  mov -0x20000(%rbp,%rax,4),%eax
0.0      _ZL12testFunctionm      4010da  jbe 401082 <_ZL12testFunctionm+0x71>
0.0      _ZL12testFunctionm      4010d0  cmpl $0x1869f,-0x20014(%rbp)
0.0      _ZL12testFunctionm      4010a0  mov -0x20010(%rbp),%eax
0.0      _ZL12testFunctionm      401094  mov -0x20000(%rbp,%rax,4),%eax
0.0      _ZL12testFunctionm      401082  movl $0x0,-0x20010(%rbp)
```

Figure 4: Our Module Output

4.2 Test 2

The following user program was profiled for number of CPU cycles taken in every function:

test2.cpp

```
void largeFunction(){
        long long n = 10000LL*100000LL;
        long long int a=1;
        for(long long i=0; i<n; i++){
                a=a*i;
        }
}

void mediumFunction(){
        long long n = 10000*10000L;
        long long int a=1;
        for(long long i=0; i<n; i++){
                a=a*i;
        }
}
```

```
void tinyFunction(long long n){
        long long n = 1000*1000L;
        long long int a=1;
        for(long long i=0; i<n; i++){
                a=a*i;
        }
}
```

The results given by perf are as follows.

```
90.81%  test2    test2            [.] _Z13largeFunctionx
 9.08%  test2    test2            [.] _Z14mediumFunctionx
 0.03%  test2    [kernel.vmlinux] [k] ioread32
 0.02%  test2    test2            [.] _Z12tinyFunctionx
 0.01%  test2    [kernel.vmlinux] [k] task_numa_work
 0.01%  test2    [kernel.vmlinux] [k] retint_user
 0.01%  test2    [kernel.vmlinux] [k] __do_softirq
 0.01%  test2    [kernel.vmlinux] [k] update_vsyscall
 0.01%  test2    [kernel.vmlinux] [k] add_interrupt_randomness
 0.01%  test2    [kernel.vmlinux] [k] do_timer
 0.01%  test2    [kernel.vmlinux] [k] __slab_free
 0.01%  test2    [kernel.vmlinux] [k] _raw_spin_lock
 0.01%  test2    [kernel.vmlinux] [k] ktime_get
 0.00%  test2    [kernel.vmlinux] [k] native_write_msr_safe
```

Figure 5: Perf Output

The results given by our module are as follows. The first column indicates the percentage of amount of CPU cycles taken by the function in the second column.

```
90.7  _Z13largeFunctionx
9     _Z14mediumFunctionx
0.1   _Z12tinyFunctionx
0     _Z27stopCountingAndGiveMeValues9pmuStruct
0     main
```

Figure 6: Our Module Output

4.3 Test 3

The following user program was profiled for number of LLC misses in every function. The `goodFunction` is supposed to perform better than `badFunction` even though they look similar. The only difference between them is that the loops are switched. For this experiment the value of `SIZE` was given as `8192`.

test3.cpp

```
float img[SIZE][SIZE];
float res[SIZE][SIZE];
void goodFunction(int SIZE){
  for(j=1;j<SIZE-1;j++) {
    for(i=1;i<SIZE-1;i++) {
            res[j][i] = 0;
            res[j][i] += img[j-1][i-1];
            res[j][i] += img[j][i-1];
            res[j][i] += img[j+1][i-1];
            res[j][i] += img[j-1][i];
            res[j][i] += img[j][i];
            res[j][i] += img[j+1][i];
            res[j][i] += img[j-1][i+1];
            res[j][i] += img[j][i+1];
            res[j][i] += img[j+1][i+1];
            res[j][i] /= 9;
        }
  }
}

void badFunction(int SIZE){
  for(i=1;i<SIZE-1;i++) {
    for(j=1;j<SIZE-1;j++) {
            res[j][i] = 0;
            res[j][i] += img[j-1][i-1];
            res[j][i] += img[j][i-1];
            res[j][i] += img[j+1][i-1];
            res[j][i] += img[j-1][i];
            res[j][i] += img[j][i];
            res[j][i] += img[j+1][i];
            res[j][i] += img[j-1][i+1];
            res[j][i] += img[j][i+1];
            res[j][i] += img[j+1][i+1];
            res[j][i] /= 9;
        }
  }
```

```
}
```

The results given by perf are as follows:

```
95.69%  test3    test3            [.] _Z11badFunctioni
 1.69%  test3    [kernel.vmlinux] [k] get_pfnblock_flags_mask
 0.47%  test3    [kernel.vmlinux] [k] clear_page_c_e
 0.17%  test3    [kernel.vmlinux] [k] page_fault
 0.17%  test3    [kernel.vmlinux] [k] _raw_spin_lock
 0.15%  test3    [kernel.vmlinux] [k] _raw_spin_trylock
 0.14%  test3    [kernel.vmlinux] [k] handle_mm_fault
 0.12%  test3    [kernel.vmlinux] [k] compact_zone
 0.12%  test3    test3            [.] _Z12goodFunctioni
```

Figure 7: Perf Output

The results given by our module are as follows:

```
96.6  _Z11badFunctioni
2.6   _Z12goodFunctioni
0     _Z27stopCountingAndGiveMeValues9pmuStruct
0     _Z27startCountingForThisProcess9pmuStruct
0     main
```

Figure 8: Our Module Output

5 Conclusion and Future Scope

The results are pretty much close to the numbers that perf gives. This validates the correctness of the tool's results. Currently our tool does not support events that only use programmable counters other than PMC0. However such an addition should be trivial. Another feature that the tool lacks is of multiplexing. That is, if you want to monitor multiple events that are measured by the same logic unit, you need to multiplex both the events and use scaling to approximate the counter values.

In this project we developed a tool that collects performance statistics of programs and compared the results of our tool with that of perf.