1 Overview

Hardware Performance Event Counters
Modern processors include features to report performance relevant events.

Events
Selected performance relevant events in processor operation, typically:
- individual instruction execution steps,
- memory cache operations,
- and many more.

Counters
Processor registers that count the occurrence of selected events.
Information collected through hardware performance event counters can help explain how the executing software interacts with the processor hardware.

2 Hardware

Hardware Architecture

Detectors
A detector is the hardware circuit responsible for reporting event occurrence.

Typical detector features:
- one detector per event type PERFEVTSEL Event Select
- configurable through additional event mask PERFEVTSEL Unit Mask
Counters
A counter is the hardware register that counts configured event occurrences.

Typical counter features:
- count in user mode or kernel mode or both
- count on single thread or on all threads across core
- count active state cycles or cycles with inactive-to-active transitions
- active state defined as event count per cycle above or below threshold
- generate interrupt on counter overflow

Architectural Performance Monitoring
Monitoring features compatible across multiple processor models.
- Standard events (clock, instructions, cache misses, branch misses)
- Standard MSR layout for configuring counters
- Fixed function performance counters

Model Specific Performance Monitoring
Monitoring features specific to single processor model.
- Model specific events (typically hundreds)
- Precise event based sampling
- Off core monitoring

Counting Speculations
Speculative execution is essential to performance. But how should speculative events be counted?

Instruction Retirement
Instruction results subject to speculation are not visible immediately. At instruction retirement phase, the processor decides whether:
- speculation was valid and results are made visible, or
- speculation was invalid and results are dropped as bogus.

Counting speculation:
- exact behavior depends on event type
  - some events inherently include speculation (branch misses)
  - events counted at retirement can typically exclude speculation
- counting at retirement introduces delay between event and count

Processor Event Based Sampling
What if more information about specific event is needed?

Record Sampling
Processor can save detailed information on event into memory buffer. Obviously this can only be done rarely or too much data is collected.

Event record contains:
- general purpose register content
- event related data (relevant address, load latency, load source)
Stored on counter overflow or on random memory load (latency sampling).

Debug Store
Event records are placed in debug store memory buffer. Interrupt is generated on buffer (near) full. Store also used for branch trace sampling.

Accessing Counters
Counter configuration done through MSR registers.
Writing
Writing requires kernel mode (WRMSR). Restricted due to security implications.

Reading
Reading typically possible from user mode (RDPMC). Features such as multiplexing or overflow handling require kernel mode anyway.

Sharing
Counters in use can be recognized in PERF_GLOBAL_INUSE MSR. Kernel context switches counters that it configured.

Example: Sandy Bridge

Numbers
3 fixed counters per thread (instructions retired and two clock types) 8 general purpose counters per core (4 per thread) 7 architectural performance events Around 200 model specific performance events 2 off core event counters (separate event types)

Features
PEBS for 4 general purpose counters and 7 event types Load latency sampling with PEBS Store destination sampling with PEBS Skid compensation for INST_RETIRED

Common Issues
Certain common issues are to be expected whenever using counters.

Inherent Properties
– Some measurement configurations not supported by hardware
– Some measurement configurations require multiplexing
  – Multiplexing can happen quite invisibly
  – Multiplexing can introduce bias
– Some sampling approaches inaccurate in principle

Wrong Usage
Relying on counter name to guess what it does is naive.

Bugs
Some processors are known to misreport values of some counters.

Example: Estimating Memory Boundedness
The goal is to determine whether workload waits for memory.

L1-DCACHE-LOADS and L1-DCACHE-LOAD-MISSES
Looks like perfect pair of counters for estimating L1 traffic?
– L1-DCACHE-LOADS counts instructions
– L1-DCACHE-LOAD-MISSES counts cache lines
– L1-DCACHE-LOAD-MISSES counts writes with RFO

MEM-LOAD-UOPS-RETIRED with various masks
Used to determine source of data in case of L1 miss?
– Prefetch counted as hit in LFB or L1
– False positives on hits in L2
– False positives on hits in L3
– Reads due to writes with RFO not counted

Based on Molka et al.: Detecting Memory Boundedness ... doi:10.1145/3030207.3030223
3 Tool: perf

Linux perf system

A framework for collecting performance related information in Linux.

Kernel Interface
- Configuration syscall
  - Selected hardware events
  - Standardized cache events
  - Low level device PMU events
  - Also software generated performance events
- Using file descriptors to operate

Userspace Tool
- Configuration
- Counting
- Sampling

Userspace perf Command

Higher-level tool to measure low-level performance in Linux. Basic features:
- Event counting
- Event-based sampling
- Supports many events:
  - Kernel provided hardware events
  - Platform specific hardware events
  - Raw event setting interface (fun to use)
  - Software events
    - perf 1ist
  - Automatic multiplexing

Gotchas

File descriptor use equals number of events times threads. Some events wrong or misleading (need to read kernel source).

perf stat command

Command to count the events of specified program.

```
perf stat [-e <EVENT> | --event=EVENT] [-a] <command>
```

Common events to specify:
- cpu-cycles/cycles
- instructions
- cache-references
- cache-misses
- branch-instructions/branches
- branch-misses
- page-faults
- context-switches/cs
- cpu-migrations/migrations

### Example Output

```bash
> perf stat <command>
```

Performance counter stats for `<command>`:

- `837.477207 task-clock (msec)` # 0.998 CPUs utilized
- `89 context-switches` # 0.106 K/sec
- `3 cpu-migrations` # 0.004 K/sec
- `2,155 page-faults` # 0.003 M/sec
- `1,974,600,340 cycles` # 2.358 GHz (83.27%)
- `1,103,816,503 stalled-cycles-frontend` # 55.90% frontend cycles idle (83.34%)
- `682,412,732 stalled-cycles-backend` # 34.56% backend cycles idle (66.74%)
- `2,124,238,728 instructions` # 1.08 insns per cycle
- `0.839128135 seconds time elapsed`

### Kernel Interface

First, opening counting mechanism, getting the file descriptor:

- `perf_event_open()` syscall

Then manipulation using:

- `ioctl()`, `prctl()` - control mechanism
- `read()`, `mmap()` - getting the data

And closing the descriptor:

- `close()`

### `perf_event_open()` Syscall

```c
#include <linux/perf_event.h>
#include <linux/hw_breakpoint.h>

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

- No library helper – call using `syscall()` function.
- For details, see manpage. Excerpts shown here.

### Interesting `perf_event_attr` fields

```c
struct perf_event_attr {
    __u32 type; /* Type of event */
    __u32 size; /* Size of structure, use sizeof */
    __u64 config; /* Type-specific configuration */

    union {
        __u64 sample_period; /* Period of sampling */
        __u64 sample_freq; /* Frequency of sampling */
    };

    __u64 sample_type; /* Specifies values included in sample */
    __u64 read_format; /* Specifies values returned in read */
    __u64 disabled : 1, /* off by default */
                  inherit : 1, /* children inherit it */
                  pinned : 1, /* must always be on PMU */
                  exclusive : 1, /* only group on PMU */
                  exclude_user : 1, /* don't count user */
                  exclude_kernel : 1, /* don't count kernel */
                  exclude_hv : 1, /* don't count hypervisor */
                  exclude_idle : 1, /* don't count when idle */
    __u64 branch_sample_type; /* enum perf_branch_sample_type */
    __u64 sample_regs_user; /* user regs to dump on samples */
    __u32 sample_stack_user; /* size of stack to dump on samples */
    ...  
```
Interesting perf_event_attr fields

Event type field
- PERF_TYPE_HARDWARE - several predefined events
- PERF_TYPE_HW_CACHE - standardized cache events
- PERF_TYPE_RAW - raw config written into event select MSR
- /sys/devices/*/type - raw config written through PMU driver

Event config field
- Specific format for each type
- Currently up to 24 bytes (config, config1, config2)
- Check /sys/devices/*/format/* for PMU driver fields
- Need to know specific system hardware

Cache Event Settings

The cache event settings are hardcoded in the kernel sources. For example, see the definition of static __initconst const u64 skl_hw_cache_event_ids in the Linux kernel sources at arch/x86/events/intel/core.c (online https://elixir.bootlin.com/linux/latest/C/ident/skl_hw_cache_event_ids):
- L1D READ OP COUNT = 0x81D0
  Event Select D0, UMask 81 = MEM_UOPS_RETIRED.ALL_LOADS
- L1D READ MISS COUNT = 0x151
  Event Select 51, UMask 01 = L1D.REPLACEMENT
  - so the first one counts instructions, the second one counts lines.

Reading counters

Counters can be read by read() syscall, returns this data:

```
struct read_format {
    u64 value; // The value of the event
    u64 time_enabled; // if PERF_FORMAT_TOTAL_TIME_ENABLED
    u64 time_running; // if PERF_FORMAT_TOTAL_TIME_RUNNING
    u64 id; // if PERF_FORMAT_ID
};
```

Reading samples

Samples can be read from a ring buffer mapped using mmap(), uses this header, data is event type specific:

```
struct perf_event_mmap_page {
    __u32 version; // version number of this structure
    __u32 compat_version; // lowest version this is compat with
    __u32 lock; // seqlock for synchronization
    __u32 index; // hardware counter identifier
    __u64 offset; // add to hardware counter value
    __u64 time_enabled; // time event active
    __u64 time_running; // time event on CPU
    __u64 capabilities; // capabilities union
    ...
    __u64 data_head; // head in the data section
    __u64 data_tail; // user-space written tail
}
```

Counter control - ioctl()

To change the status of counter, we can call ioctl(). Some interesting commands:
- PERF_EVENT_IOC_ENABLE
- PERF_EVENT_IOC_DISABLE
- PERF_EVENT_IOC_RESET

More perf Examples

The examples use the prime sieve example from https://github.com/d-iii-s/teaching-performance-evaluation/tree/master/src/experiment-prime-sieve
# Provide variability indication for instruction count.
perf stat -e instructions -r 5 ./checked-smart

# Count in user mode only.
perf stat -e instructions:u -r 5 ./checked-smart
# Count in kernel mode only.
perf stat -e instructions:k -r 5 ./checked-smart

# See how counter multiplexing interpolates results.
# Increase the count of instruction events requested
# until interpolated results are printed.
perf stat -e instructions -r 5 ./checked-smart
perf stat -e instructions,instructions -r 5 ./checked-smart
perf stat -e instructions,instructions,instructions -r 5 ./checked-smart ...

# Counter groups are measured together even with counter multiplexing.
perf stat -e '{instructions,instructions}' -r 5 ./checked-smart

# Count across all processes while a command is running.
# May require administrative privileges.
perf stat -a -e instructions -r 5 ./checked-smart

# Report results for each processor separately.
# May require administrative privileges.
perf stat -A -a -e instructions -r 5 ./checked-smart

# Record and report memory access information.
perf mem record ./basic
perf mem report

# Attach to a running process.
perf stat -p <PID> ...

# Display interactive results.
perf top

## 4 Tool: PAPI

**PAPI**

PAPI is a standard library that provides application level interface to performance counters on multiple hardware and operating system platforms.

PAPI does not contain code to access performance counters, support from operating system kernel is required.

Main features of PAPI are:
- per thread virtual performance counters,
- support for platform independent events,
- support for counter multiplexing.

[http://icl.cs.utk.edu/papi](http://icl.cs.utk.edu/papi)

**Programming with PAPI**

**Concepts**

**Native event** is a performance event supported on a specific platform.
**Preset event** is a standardized performance event that the library supports on multiple platforms. Some preset events are derived from multiple native events.

**Event set** is a set of events that are monitored together.

Example simplified from PAPI distribution. Error checking omitted for brevity.

```c
// Initialize the library.
PAPI_library_init (PAPI_VER_CURRENT);

// Define an event set to be used.
int events = PAPI_NULL;
PAPI_create_eventset (&events);
PAPI_add_event (events, PAPI_TOT_INS);
PAPI_add_event (events, PAPI_TOT_CYC);

// Start counting using the event set.
long long values [2];
PAPI_start (events);
...
PAPI_read (events, values);
...
PAPI_stop (events, values);

...

// Define an event set with a native event.
int events = PAPI_NULL;
PAPI_create_eventset (&events);
int native;
PAPI_event_name_to_code ("RESOURCE_STALLS:ANY", &native);
PAPI_add_event (events, native);
...

// Initialize the library.
PAPI_library_init (PAPI_VER_CURRENT);
PAPI_multiplex_init ();

// Create event set with multiplex flag.
PAPI_create_eventset (&events);
PAPI_assign_eventset_component (events, 0);
PAPI_set_multiplex (events);
...

// Close the library.
PAPI_cleanup_eventset (events);
PAPI_destroy_eventset (&events);
PAPI_shutdown ();
```

## 5 Tool: LIKWID

**Hardware Topology Information**

```bash
> likwid-topology -g
Socket #:
```

---
### Performance Counter Groups

> likwid-perfctr --a

<table>
<thead>
<tr>
<th>Branch</th>
<th>Branch prediction miss rate/ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Caches</td>
<td>Some data from the CBOXes</td>
</tr>
<tr>
<td>Clock</td>
<td>Power and Energy consumption</td>
</tr>
<tr>
<td>Energy</td>
<td>Power and Energy consumption</td>
</tr>
<tr>
<td>Mem</td>
<td>Main memory bandwidth in MBytes/s</td>
</tr>
<tr>
<td>Numa</td>
<td>Local and remote memory accesses</td>
</tr>
<tr>
<td>Qpi</td>
<td>Qpi traffic between sockets</td>
</tr>
</tbody>
</table>

> likwid-perfctr -H -g CACHES

#### CACHES

Some data from the CBOXes

- **Branch**
  - Branch prediction miss rate/ratio
- **Caches**
  - Some data from the CBOXes
- **Clock**
  - Power and Energy consumption
- **Energy**
  - Power and Energy consumption
- **Mem**
  - Main memory bandwidth in MBytes/s
- **Numa**
  - Local and remote memory accesses
- **Qpi**
  - Qpi traffic between sockets

---

### Power Consumption Measurement

> likwid-powermeter -p <command>

---

#### Power Consumption Measurement

<table>
<thead>
<tr>
<th>Event</th>
<th>Counter</th>
<th>Sum</th>
<th>Min</th>
<th>Max</th>
<th>Avg</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Instr-retired:**
  - Instruction retirement
- **Cpu_clk_unhalted-core:**
  - Core clock unhalted
- **Cpu_clk_unhalted-ref:**
  - Reference clock unhalted
- **Pwr_mng_energy:**
  - Power management energy
- **Par dram_energy:**
  - Parallel dram energy

---

<table>
<thead>
<tr>
<th>Metric</th>
<th>Counters</th>
<th>Sum</th>
<th>Min</th>
<th>Max</th>
<th>Avg</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Runtime (RDTS)**
  - Time since boot
- **Runtime unhalted:**
  - Time since boot (unhalted)
- **Clock (MHz):**
  - Clock frequency in MHz
- **Energy:**
  - Total energy consumption
- **Energy [J]:**
  - Energy consumption in Joules
- **Power:**
  - Power consumption
- **Power [W]:**
  - Power consumption in Watts

---

|        |         |       |       |       |       |
|        |          |       |       |       |       |
6 Analysis

Example: Top Down Analysis

The goal is to identify major performance bottlenecks.

Analysis Hierarchy

Defined to reflect modern processor architecture. Classifies individual pipeline slots.

- Frontend bound slots Slots where no μops are issued despite backend not stalled
- Backend bound slots Slots where no μops are issued because backend is stalled
- Bad speculation slots Slots with μops that were eventually discarded as bogus
- Retirement slots Slots with μops that retire normally (this is good)

Further hierarchy steps for each branch follow.

The example is based on the paper by on Yasin: A Top Down Method ... doi:10.1109/ISPASS.2014.6844459

Examine Figure 1 for details on the general processor architecture. Examine Figure 2 for the hierarchy of slot classifications. Figure 4 provides an example flow chart to classify memory related stalls in backend bound slots. Appendix I gives the specific hardware performance counters used to apply the slot classification.

Looking at Figures 5 and 6 gives an impression of what kind of information the analysis delivers. The figures include mostly backend bound workloads, some information can also be gleaned from comparing different parallelism levels (a shift from cache bound to memory bound slots indicates potential cache capacity issue).

Figure 8 demonstrates how differences in processor architecture are reflected on the classification. Also fun to note is the fact that some figures contain negative values, due to counter errata.

The top down analysis is supported by several existing software tools, including (partially) the 'perf' tool:

```
perf stat --topdown --all-cpus ./basic
```

---

Based on Yasin: A Top Down Method ... doi:10.1109/ISPASS.2014.6844459