Self-monitoring Overhead of the Linux perf_event Performance Counter Interface

Vince Weaver
University of Maine
vincent.weaver@maine.edu

ISPASS 2015 – 30 March 2015
Hardware Performance Counters

• Low-level CPU registers that measure architectural events (cycles, instructions, cache misses, branch misses, memory accesses, estimated power)

• Found on most modern CPUs, including all x86 and most ARM
Linux and Performance Counters

• Linux – operating system used everywhere, from embedded phones to top500 supercomputers

• Until Linux 2.6.31 (2009) no support for performance counters; perfctr and perfmon2 required kernel patches.
Linux perf_event

- A lot of time was wasted trying to get perfmon2 merged.
- Meanwhile Molnar et al. implemented perf_event interface from scratch and quickly got it merged.
- It took a few years, but perf_event now is mostly feature complete, though it sometimes lags a bit with new CPU releases (especially some of the esoteric new monitoring features from Intel)
perf_event Interface

- Very complex interface. The `perf_event_open()` system call has 40+ parameters. It currently has the longest manpage of any syscall.

- Governing philosophy: do everything in the kernel.

- Most usage patterns are to open an event, then use common calls like `read()`, `ioctl()`, `poll()` and `mmap()` to gather results.
What is the Overhead of the Interface?

• Overhead of the operating system interface.

• The overhead from enabling the hardware is usually considered to be zero.

• Compare `perf_event` against `perfctr` and `permon2`
Performance Counter Usage

• Aggregate Counts – total for entire run of a program
  low overhead, low detail

• Sampled Execution – execution periodically interrupted
  and stats logged for later analysis
  variable overhead, medium detail

• Self Monitoring – calipers around exact code of interest
  unknown overhead, high detail
Self Monitoring

- Used by PAPI (Performance API), not perf

- Sample code

```c
/* Event opened in advance with perf_event_open() */

/* start measurement */
ioctl(fd, PERF_EVENT_IOC_ENABLE, 0);

/* stop measurement */
ioctl(fd, PERF_EVENT_IOC_DISABLE, 0);

/* read results */
read(fd, buffer, BUFFER_SIZE*sizeof(long long));
```

CODE OF INTEREST
## Machines Investigated

<table>
<thead>
<tr>
<th>Processor</th>
<th>Counters Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Atom Cedarview D2550</td>
<td>2 general 3 fixed</td>
</tr>
<tr>
<td>Intel Core2 P8700</td>
<td>2 general 3 fixed</td>
</tr>
<tr>
<td>Intel IvyBridge i5-3210M</td>
<td>4 general 3 fixed</td>
</tr>
<tr>
<td>AMD Bobcat G-T56N</td>
<td>4 general</td>
</tr>
</tbody>
</table>
Methodology

• Use rdtsc timestamp counter to measure overhead

• Disable DVFS frequency scaling

• Use same version of gcc (4.4) to compile all the kernels

• Code of interest is empty to avoid that affecting results (start/stop/read with nothing intervening)

• Run test 1024 times, show boxplots
Compiler effect on Kernel

Average Overhead (Cycles)

core2 Overhead of Read with 1 Event

gcc 4.4  gcc 4.5  gcc 4.6  gcc 4.7  gcc 4.8

0 1000 2000 3000

n/a
Overhead Total (core2)

Overall Start/Stop/Read Overhead with 1 Event (core2)
Overhead Start/Stop

Start Overhead with 1 Event (core2)

Stop Overhead with 1 Event (core2)
What about using `rdpmc`?
Why are reads slow?

• Dynamic vs Static linking (first call to read)

• `rdpmc` – first access to mmap page causes pagefault. `perfctr` avoids this, pre-faults the page. For `perf_event` we can touch the page or use `MAP_POPULATE`. 
Updated Read Overheads Core2

core2 Overhead of Read with 1 Event

Average Overhead (Cycles)

0 1000 2000 3000 4000

3.16
3.16-syscall
3.16-syscall_static
3.16-rdpmc
3.16-rdpmc-populate
3.16-rdpmc-touch
3.16-rdpmc-touch_static
2.6.30-perfmon2
2.6.32-perfctr
Updated Read Overheads All

- **core2 Overhead of Read with 1 Event**
- **bobcat Overhead of Read with 1 Event**
- **ivybridge Overhead of Read with 1 Event**
- **atom-cedarview Overhead of Read with 1 Event**
Overhead Mitigated by Successive Reads?

![Graph showing core2 Overhead of Successive Read](image-url)

- Average Overhead (Cycles)
- Successive Counter Reads
- core2 Overhead of Successive Read
- rdpmc-touch-static
- 3.18
Seems to be a Cache Issue

![core2 Overhead of Successive Read](chart)

- **Average Overhead (Cycles)**
- **Successive Counter Reads**
- **Core2 Overhead of Successive Read**
- **3.18**
- **rdpmc-touch-static**
- **L1D$ miss x100**

---

19
rdpmc Results as Expected

![Graph showing core2 Overhead of Successive Read](image)
Scaling as we read Multiple Counters

Simultaneous Events Being Measured

core2 Overall Overhead

Average Time (Cycles)

Read Time
Stop Time
Start Time

3.18
3.18
3.18
static
static,touch
Perfmon2
Perfctr

Simultaneous Events Being Measured
Conclusions

• The default self-monitoring overhead of perf_event is high, but it can be mitigated.

• Read overhead can be vastly improved with proper setup.

• Start and stop overhead is higher than other implementations, but this is likely due to limitations of the interface.
Future Work

• Modify PAPI to use the improved `rdpmc` interface

• Explore non-x86 architectures

• Investigate overhead of aggregate and sampled methodologies
Questions?

vincent.weaver@maine.edu

All code and data is available

http://web.eece.maine.edu/~vweaver/projects/perf_events/overhead

    git://github.com/deater/perfevent_overhead.git