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.
  `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 perfmon2

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

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

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

CODE OF INTEREST

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

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

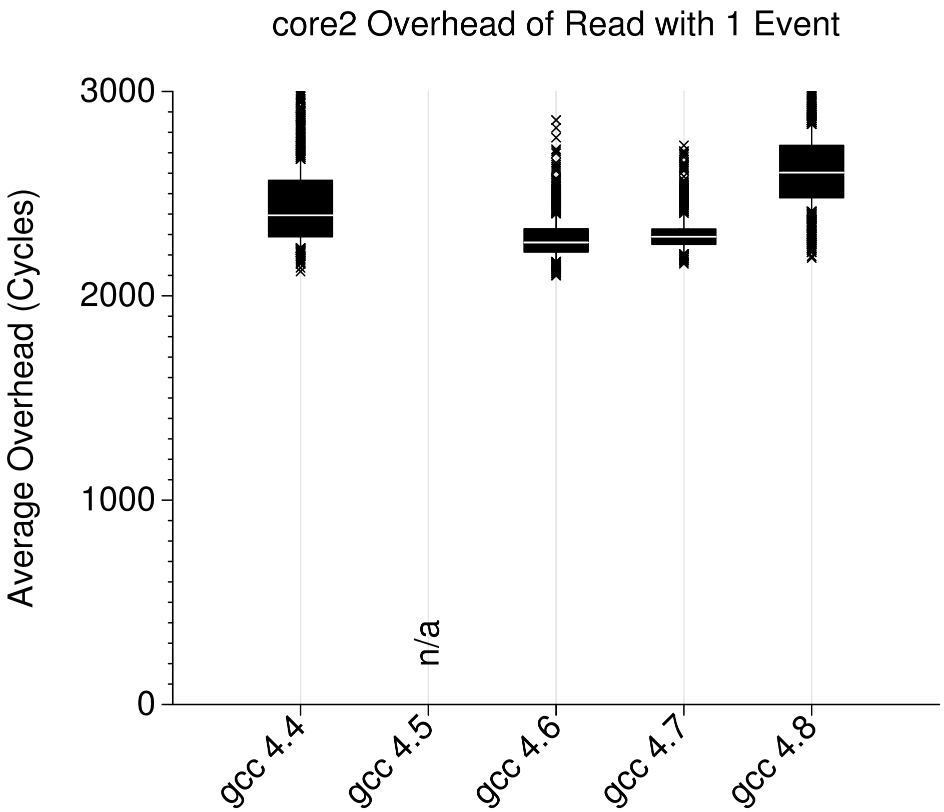
Machines Investigated

Processor	Counters Available
Intel Atom Cedarview D2550	2 general 3 fixed
Intel Core2 P8700	2 general 3 fixed
Intel IvyBridge i5-3210M	4 general 3 fixed
AMD Bobcat G-T56N	4 general

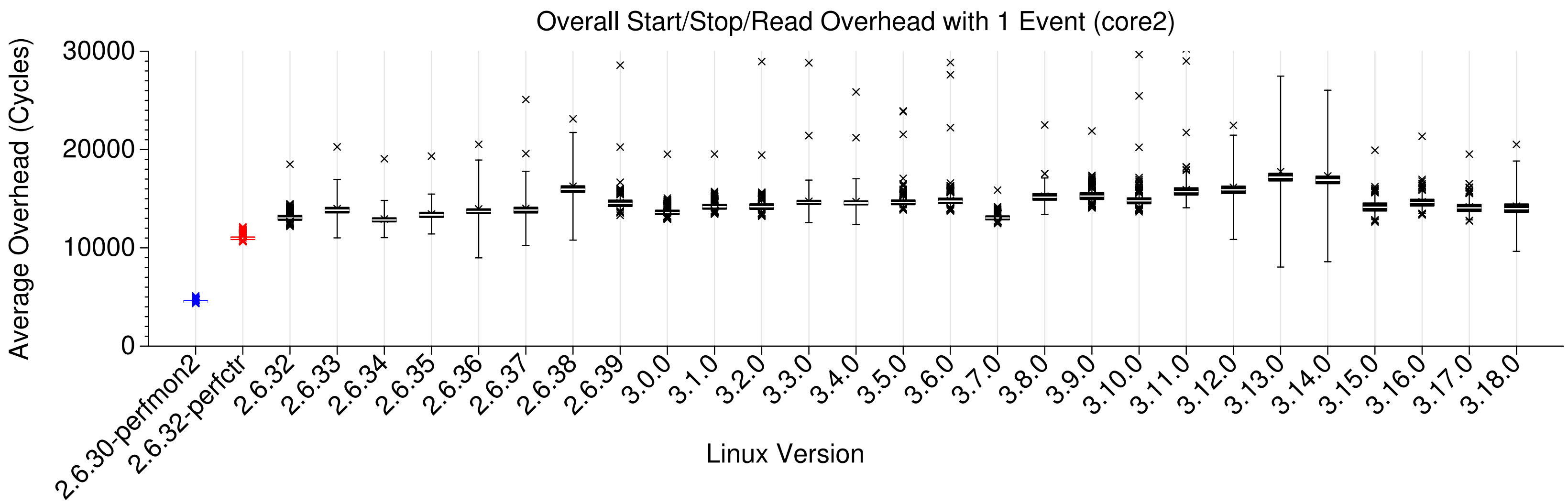
Methodology

- 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

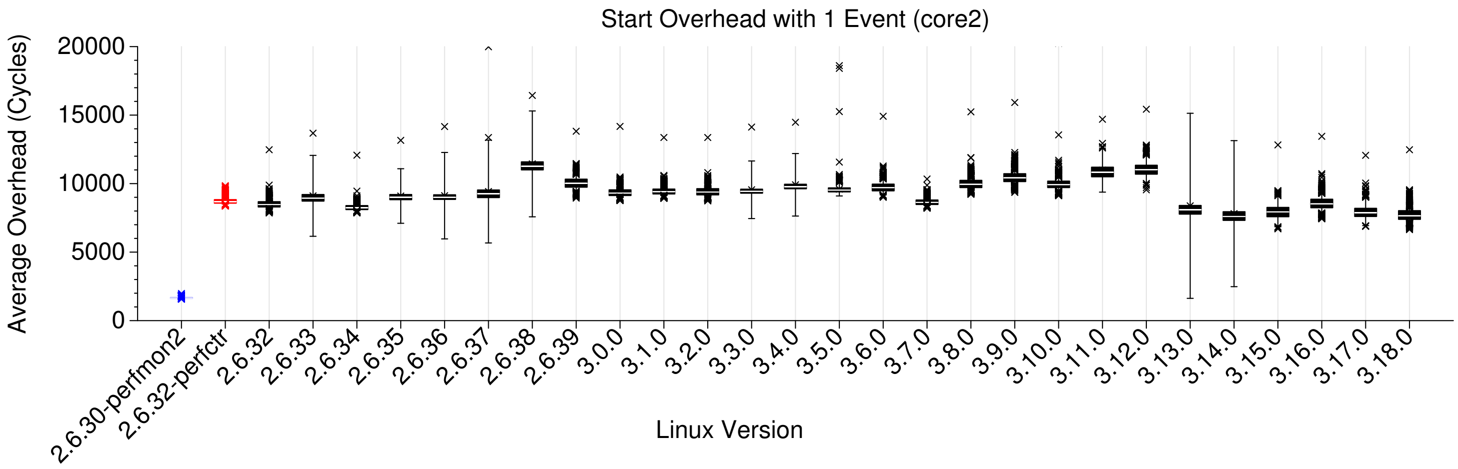
Compiler effect on Kernel

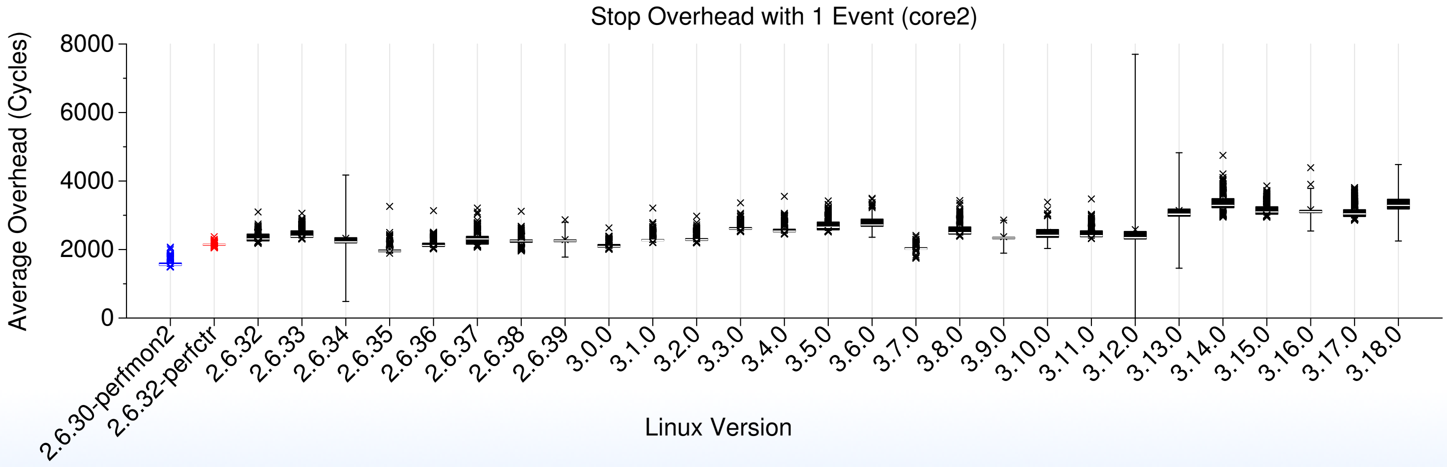


Overhead Total (core2)

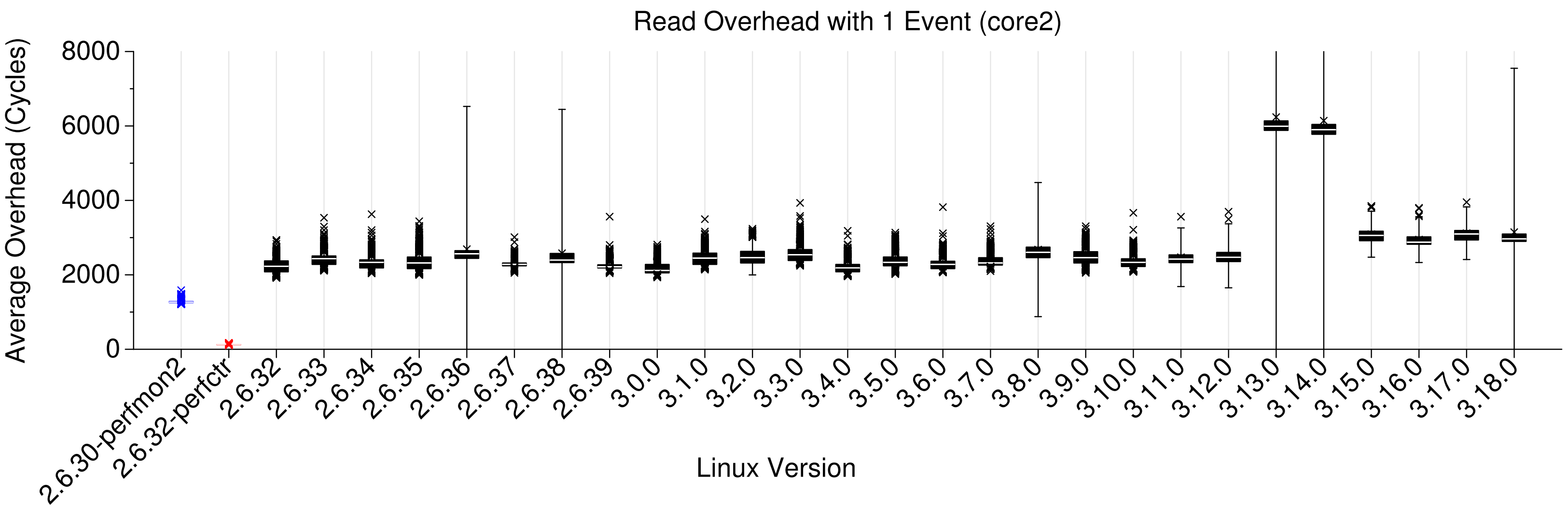


Overhead Start/Stop

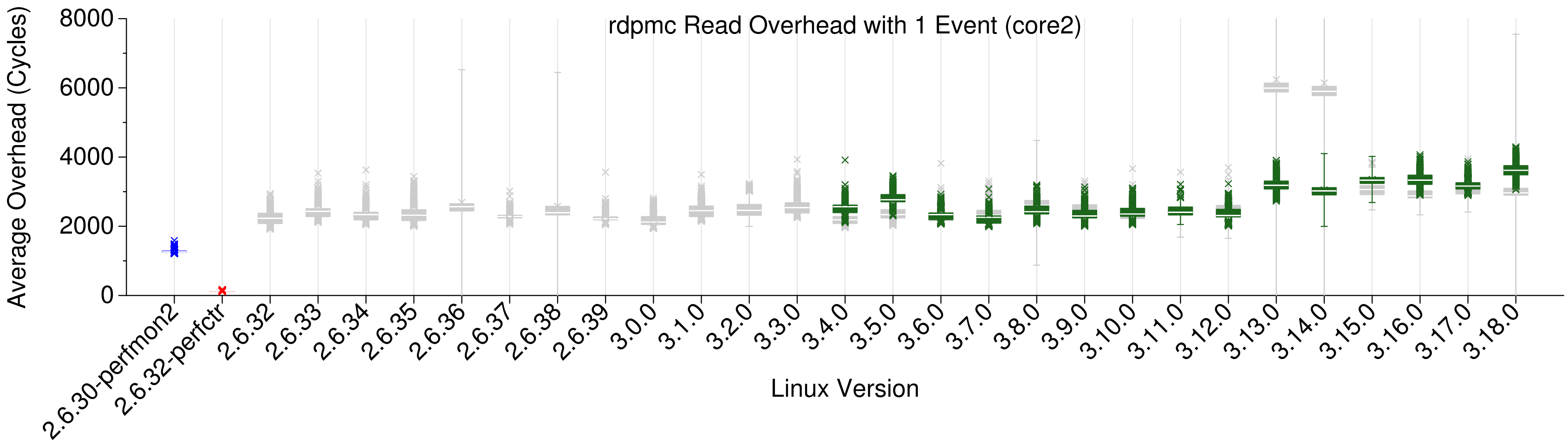

Start Overhead with 1 Event (core2)


Stop Overhead with 1 Event (core2)

Overhead Read



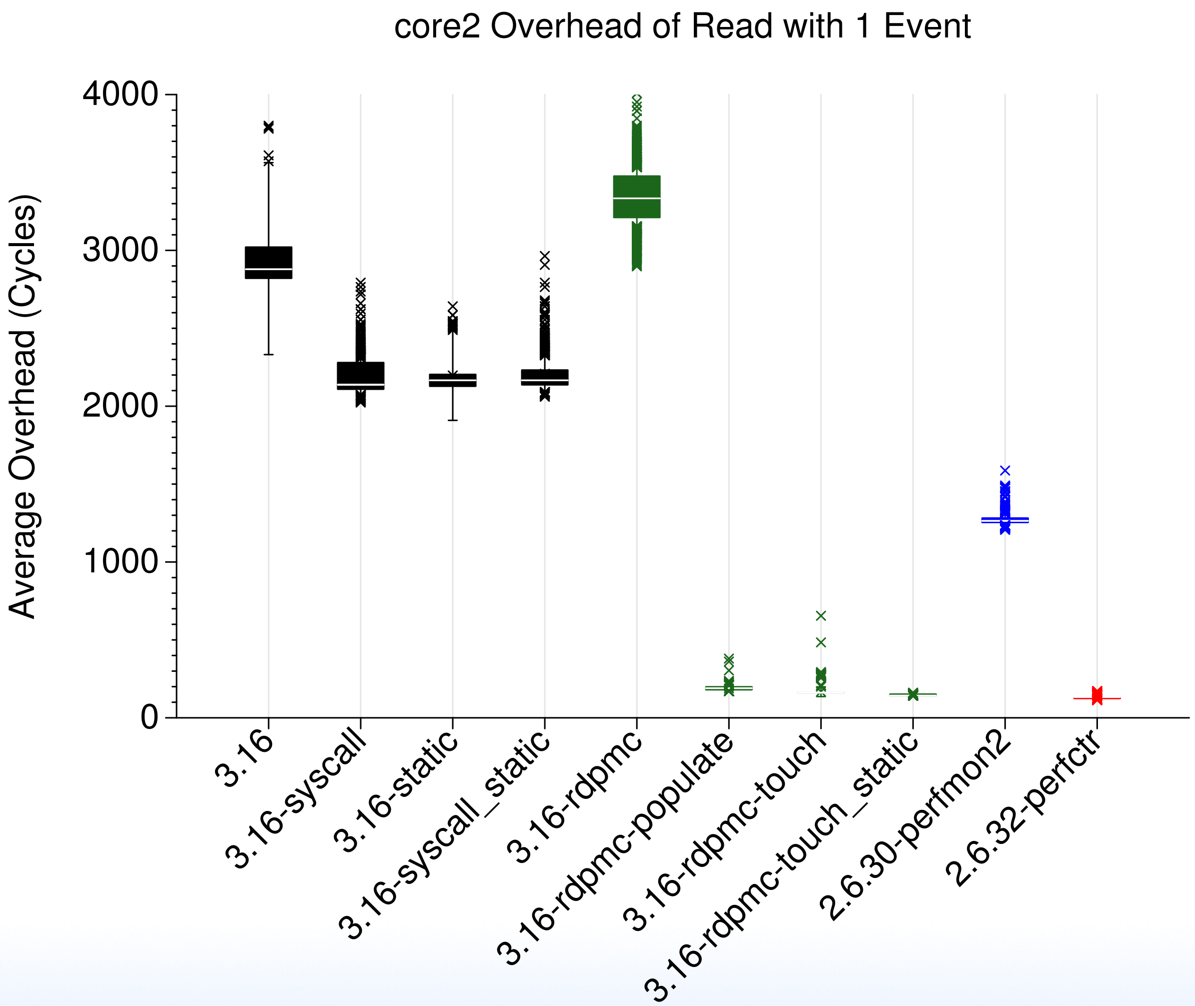
What about using `rdpmc`?



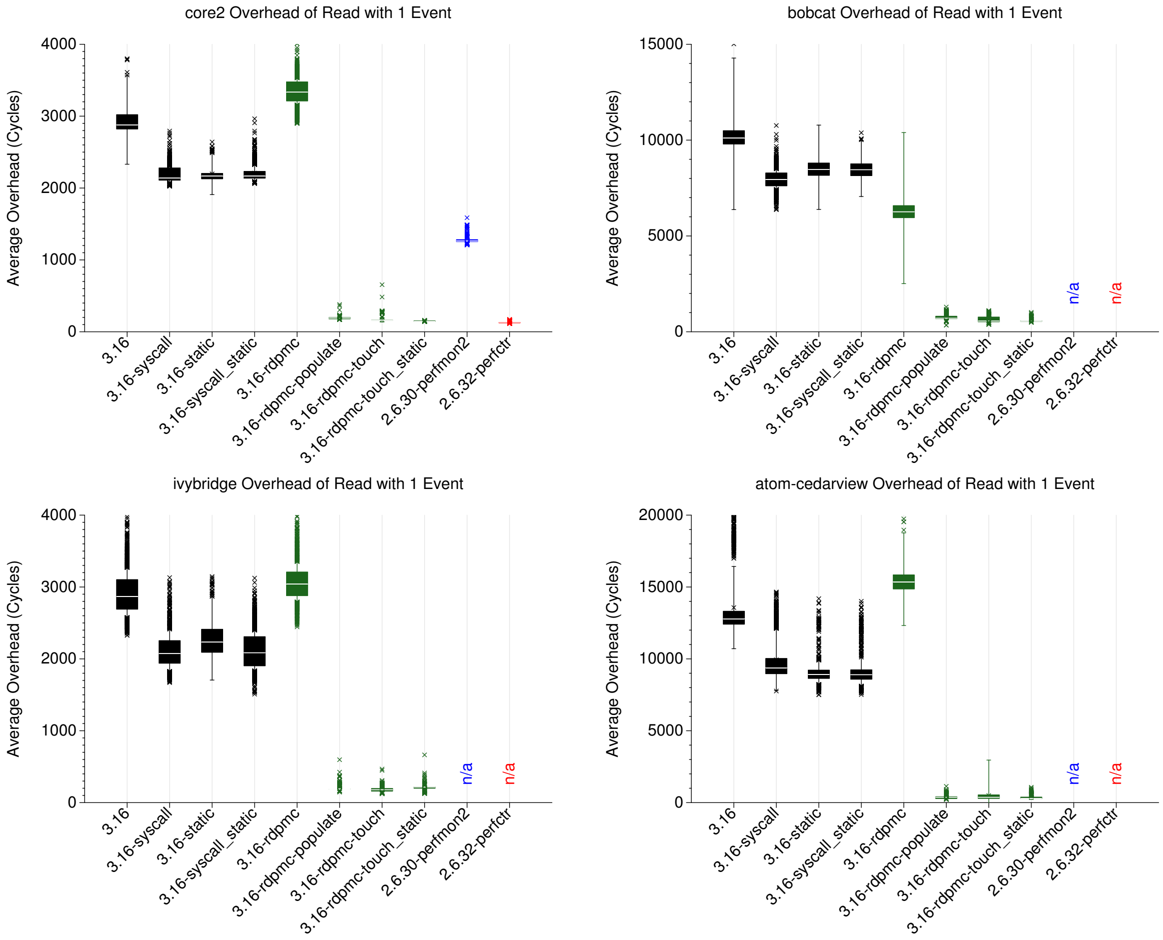
Why are reads slow?

- 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`.

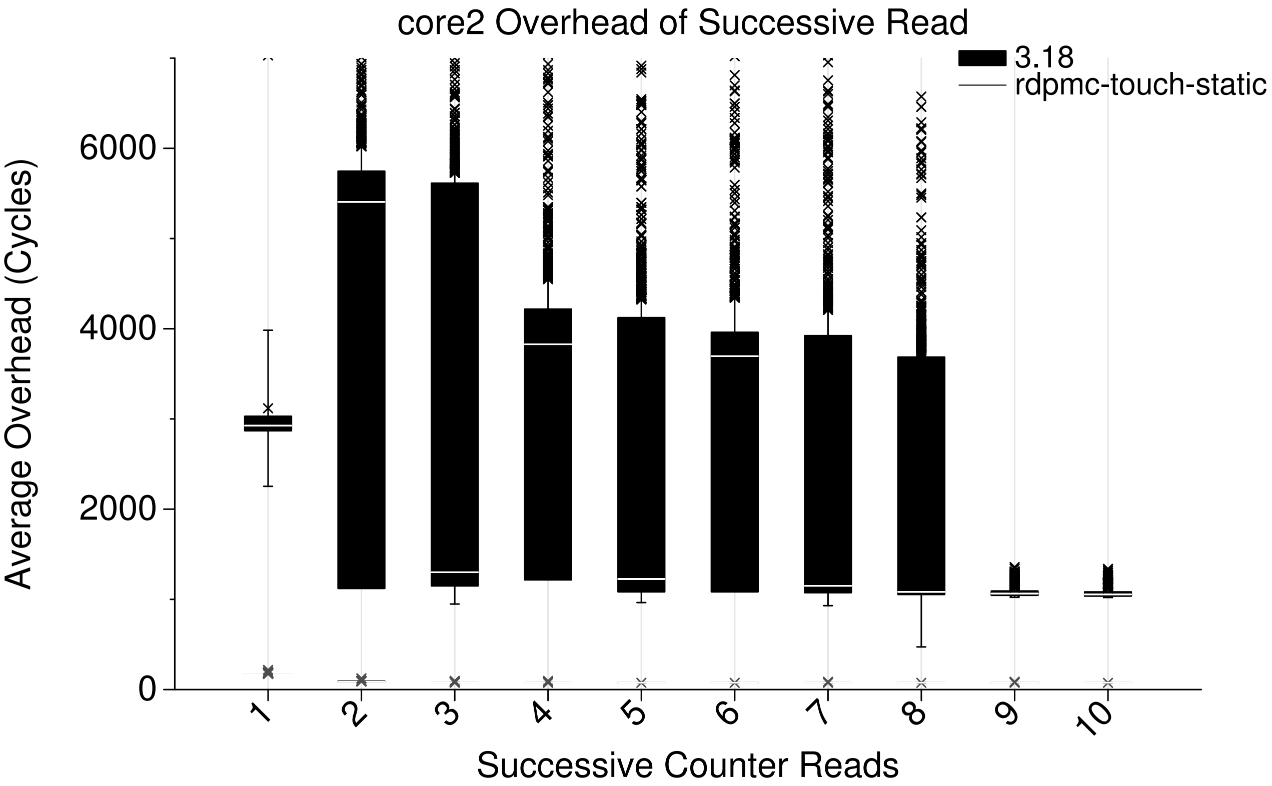
Updated Read Overheads Core2



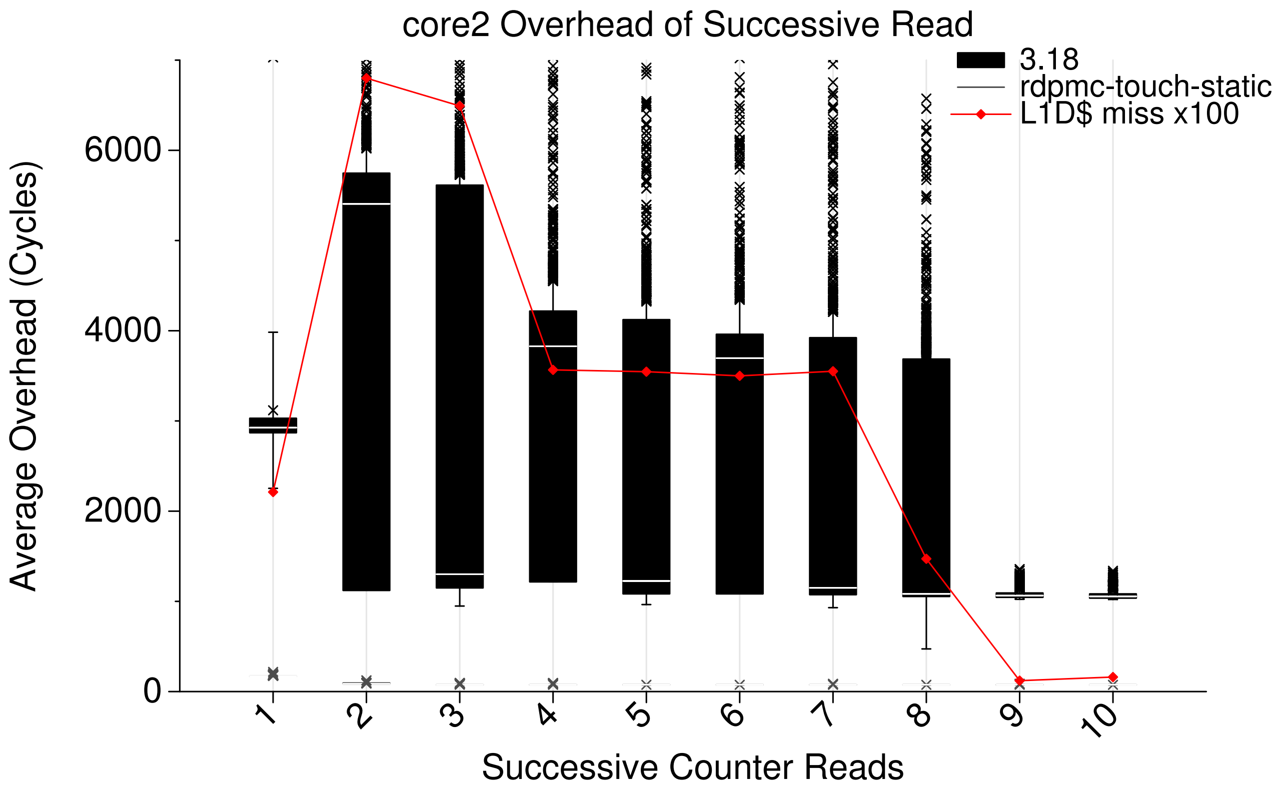
Updated Read Overheads All



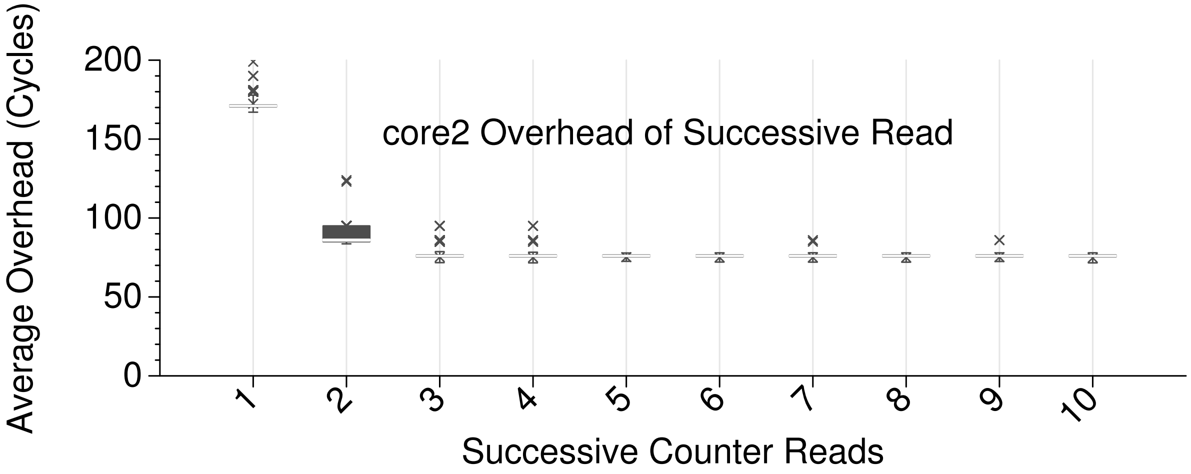
Overhead Mitigated by Successive Reads?



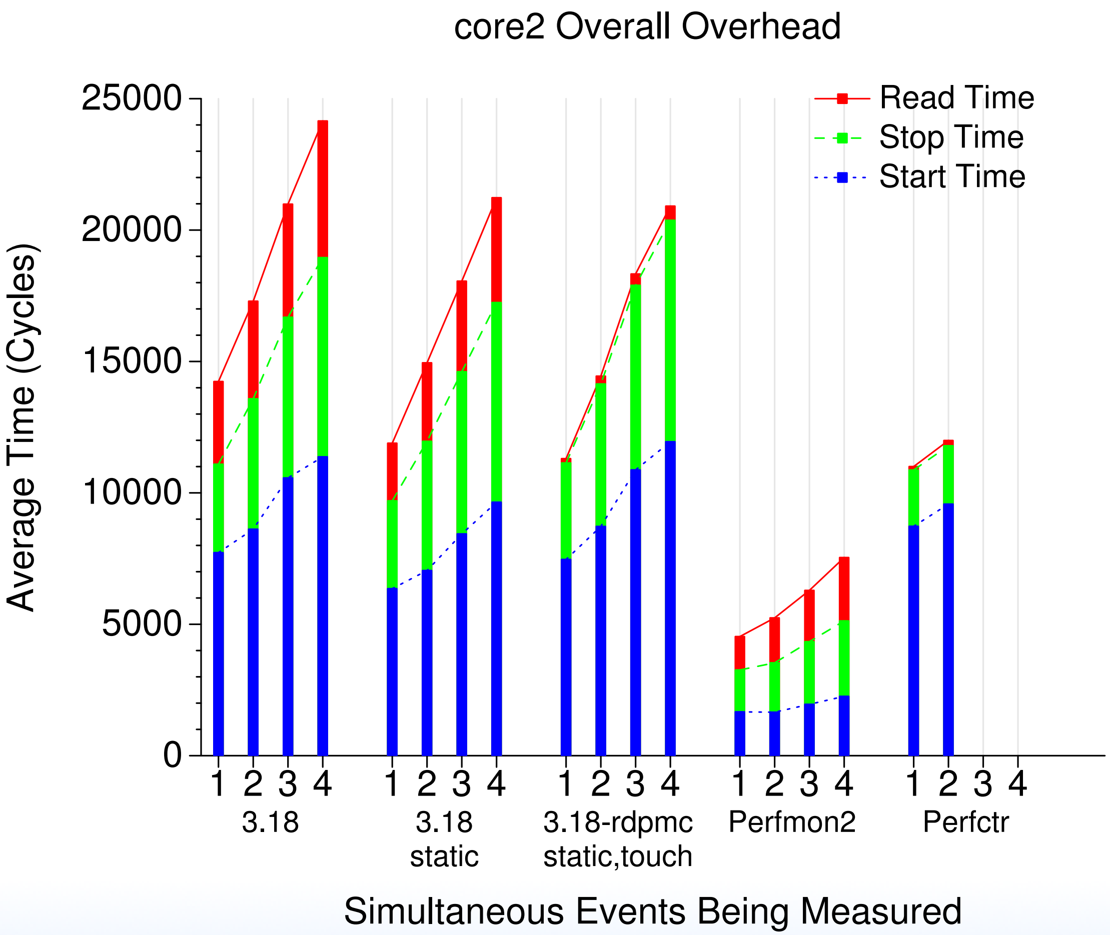
Seems to be a Cache Issue



`rdpmc` Results as Expected



Scaling as we read Multiple Counters



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