Linux perf_event Features and Overhead

2013 FastPath Workshop

Vince Weaver

http://www.eece.maine.edu/~vweaver

vincent.weaver@maine.edu

21 April 2013

Performance Counters and Workload Optimized Systems

- With processor speeds constant, cannot depend on Moore's Law to deliver increased performance
- Code analysis and optimization can provide speedups in existing code on existing hardware
- Systems with a single workload are best target for cross-stack hardware/kernel/application optimization
- Hardware performance counters are the perfect tool for this type of optimization

Some Uses of Performance Counters

- Traditional analysis and optimization
- Finding architectural reasons for slowdown
- Validating Simulators
- Auto-tuning
- Operating System optimization
- Estimating power/energy in software

Linux and Performance Counters

- Linux has become the operating system of choice in many domains
- Runs most of the Top500 list (over 90%) on down to embedded devices (Android Phones)
- Until recently had no easy access to hardware performance counters, limiting code analysis and optimization.

Linux Performance Counter History

- oprofile – system-wide sampling profiler since 2002
- perfctr – widely used general interface available since 1999, required patching kernel
- perfmon2 – another general interface, included in kernel for itanium, made generic, big push for kernel inclusion

Linux perf_event

- Developed in response to perfmon2 by Molnar and Gleixner in 2009
- Merged in 2.6.31 as “PCL”
- Unusual design pushes most functionality into kernel
- Not well documented nor well characterized

perf_event Interface

- `sys_perf_event_open()` system call
- complex `perf_event_attr` structure (over 40 fields)
- counters started/stopped with `ioctl()` call
- values read either with `read()` or samples in `mmap()` circular buffer

perf_event Kernel Features

- Generalized Events – commonly used events on various architectures provided common names
- Event Scheduling – kernel handles mapping events to appropriate counters
- Multiplexing – if more events than counters, time based multiplexing extrapolates full counts
- Per-process counts – values saved on context switch
- Software Events – kernel events exposed by same API

Advanced Hardware Features

- Offcore Response – filtered measuring of memory accesses that leave the core
- Uncore and Northbridge Events – special support needed for shared resources (L2, L3, memory)
- Sampled Interfaces
  - + AMD Instruction Based Sampling (IBS) – can provide address, latency, etc., as well as minimal skid
  - + Intel Precise Event Sampling (PEBS) – gathers extra data on triggered event (registers, latency), low-skid

Virtualized Counters

- Recent versions of KVM can trap on access to performance MSRs and pass in guest-specific performance counts, allowing use of performance counters in a virtualized environment
- counter values have to be save/restored when guest scheduled

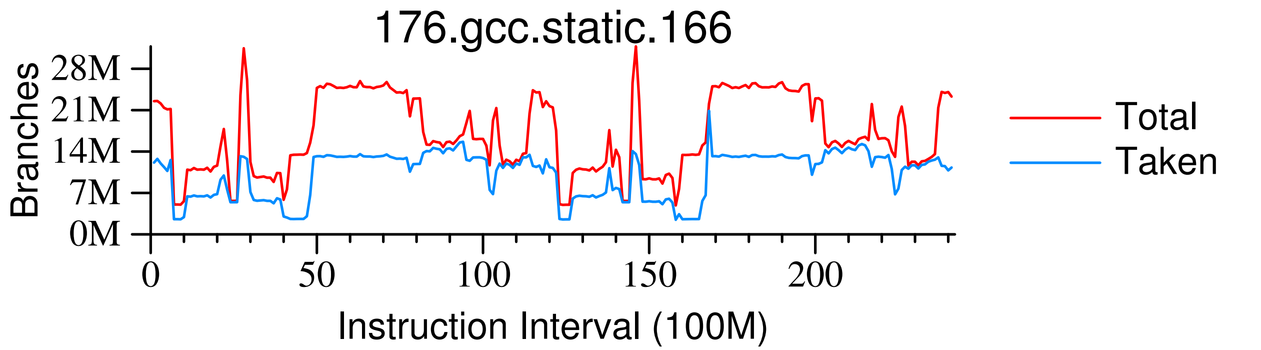
More on Generalized Events

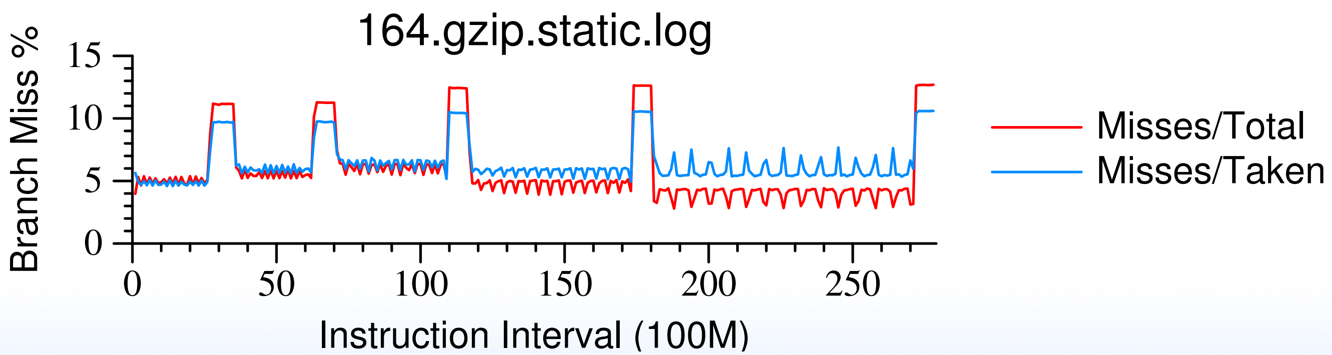
- Unlike those provided by user-space libraries (PAPI), hard to know what the actual event is (this is changing)

- Kernel events are sometimes wrong, a lot more hassle to update kernel than update library

Generalized Events – Wrong Events

Until 2.6.35 total "branches" preset accidentally mapped to "taken branches"

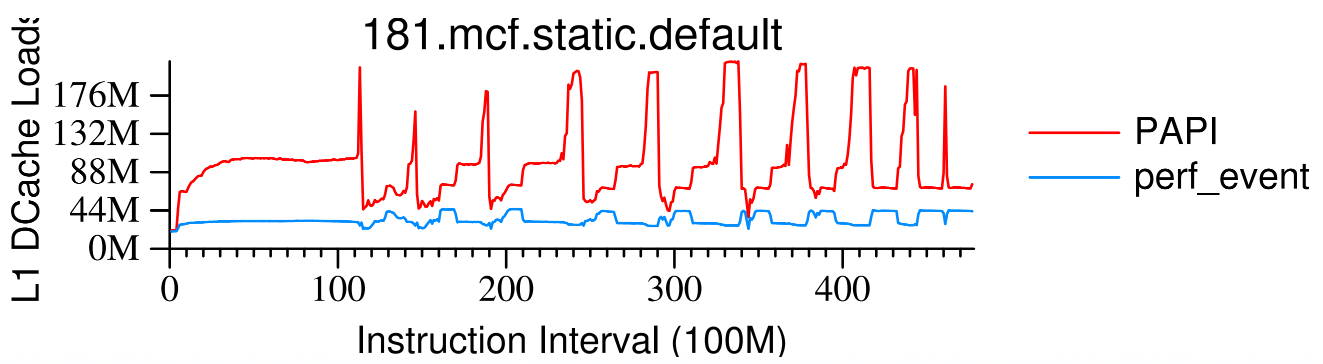




Generalized Events – Similar Events, Different Meaning

On Nehalem,

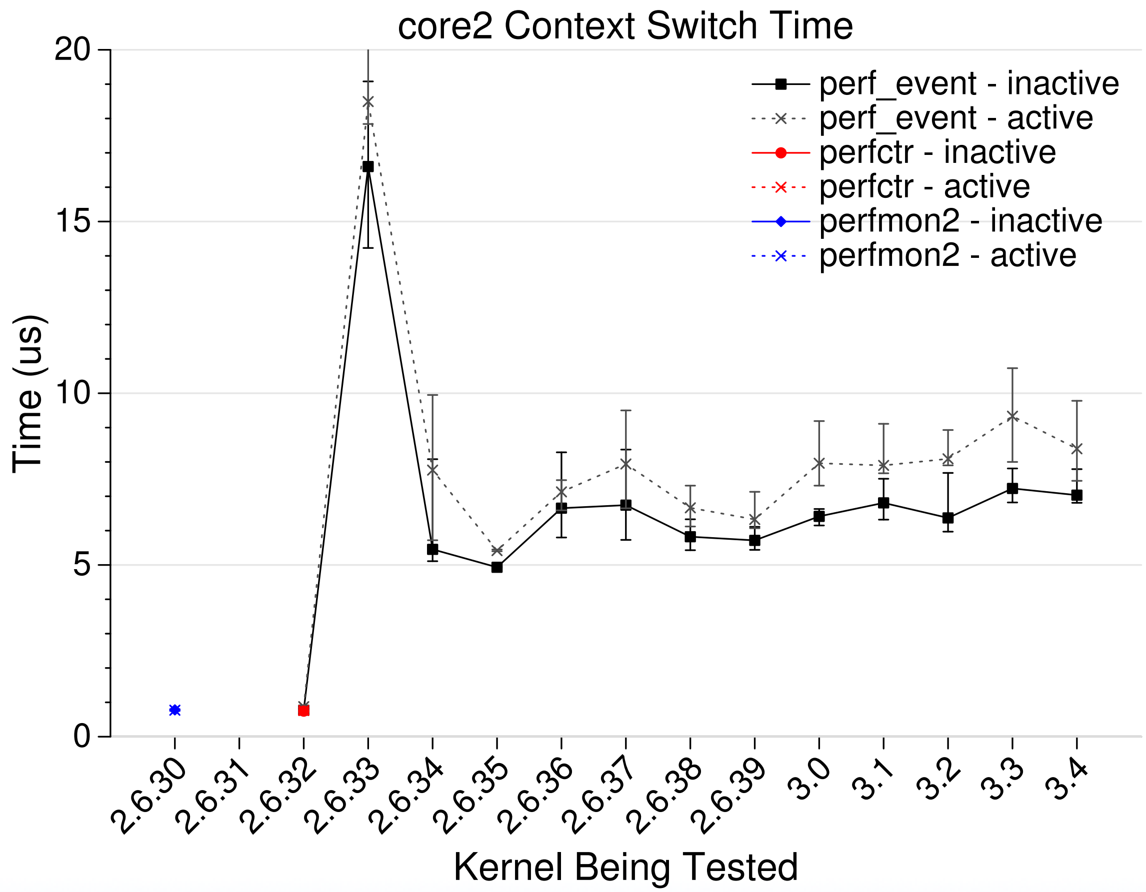
- perf_event defines `L1D.OP_READ.RESULT_ACCESS` (perf: L1-dcache-loads) as `MEM_INT_RETIRED:LOADS`
- PAPI defines PAPI_L1_DCR as `L1D_CACHE_LD:MESI`



Context-Switch Test Methodology

- To give per-process events, have to save counts on context-switch. This has overhead
- We use `lmbench` `lat_ctx` benchmark. Run it with and without `perf` measuring it.
- Up to 20% overhead when perf monitoring the threads. Benchmark documentation claim 10-15% accuracy at best

Core2 Context-Switch Overhead



Common Performance Counter Usage Models

- Aggregate
- Sampled
- Self-monitoring

Linux perf_event can do all three.

Aggregate Counts

```
$ perf stat -e instructions,cycles,branches,branch-misses,cache-misses
./matrix_multiply_atlas
Matrix multiply sum: s=3650244631906855424.000000

 Performance counter stats for './matrix_multiply_atlas':

   194,492,378,876 instructions              #    2.51  insns per cycle
    77,585,141,514 cycles                    #    0.000 GHz
       584,202,927 branches
         3,963,325 branch-misses             #    0.68% of all branches
        89,863,007 cache-misses

      49.973787489 seconds time elapsed
```

perf_event sets up events, forks process (start counts on `exec()`), handles overflow, waits for exit, prints totals.

Sampled Profiling

```
$ perf record ./matrix_multiply_atlas
Matrix multiply sum: s=3650244631906855424.000000
[ perf record: Woken up 14 times to write data ]
[ perf record: Captured and wrote 3.757 MB perf.data (~164126 samples) ]
$ perf report
Events: 98K cycles
 97.36%  matrix_multiply  libblas.so.3.0         [.] ATL_dJIK48x48x48TN48x
  0.62%  matrix_multiply  matrix_multiply_atlas  [.] naive_matrix_multiply
  0.27%  matrix_multiply  libblas.so.3.0         [.] 0x1f1728
  0.18%  matrix_multiply  libblas.so.3.0         [.] ATL_dupMBmm0_8_0_b1
  0.16%  matrix_multiply  libblas.so.3.0         [.] ATL_dupKBmm8_2_1_b1
  0.14%  matrix_multiply  libblas.so.3.0         [.] ATL_dupNBmm0_1_0_b1
  0.13%  matrix_multiply  libblas.so.3.0         [.] ATL_dcol2blk_a1
  0.09%  matrix_multiply  [kernel.kallsyms]      [k] page_fault
```

Periodically sample, grad state, record for later analysis.

Self-Monitoring

```
retval = PAPI_library_init(PAPI_VER_CURRENT);
if (retval != PAPI_VER_CURRENT) fprintf(stderr,"Wrong PAPI version\n");

retval = PAPI_create_eventset( &event_set);
if (retval != PAPI_OK) fprintf(stderr,"Error creating eventset\n");

retval = PAPI_add_named_event( event_set, "PAPI_TOT_INS" );
if (retval != PAPI_OK) fprintf(stderr,"Error adding event\n");

retval = PAPI_start(event_set);

naive_matrix_multiply(0);

retval = PAPI_stop(event_set,&count);

printf("Total instructions: %lld\n",count);
```

Self-Monitoring Overhead

- Typical pattern is Start/Stop/Read
- Want minimal possible overhead
- Read performance is typically most important, especially if doing multiple reads

Methodology

- DVFS disabled
- Use rdtsc() 64-bit timestamp counter. Typically 150 cycle overhead
- Measure start/stop/read with no code in between
- All three (start/stop/read) measured at same time
- Environment variables should not matter

perf_event Measurement Code

```
start_before=rdtsc();

ioctl(fd[0], PERF_EVENT_IOC_ENABLE,0);

start_after=rdtsc();

ioctl(fd[0], PERF_EVENT_IOC_DISABLE,0);

stop_after=rdtsc();

read(fd[0],buffer,BUFFER_SIZE*sizeof(long long));

read_after=rdtsc();
```

perfctr Measurement Code

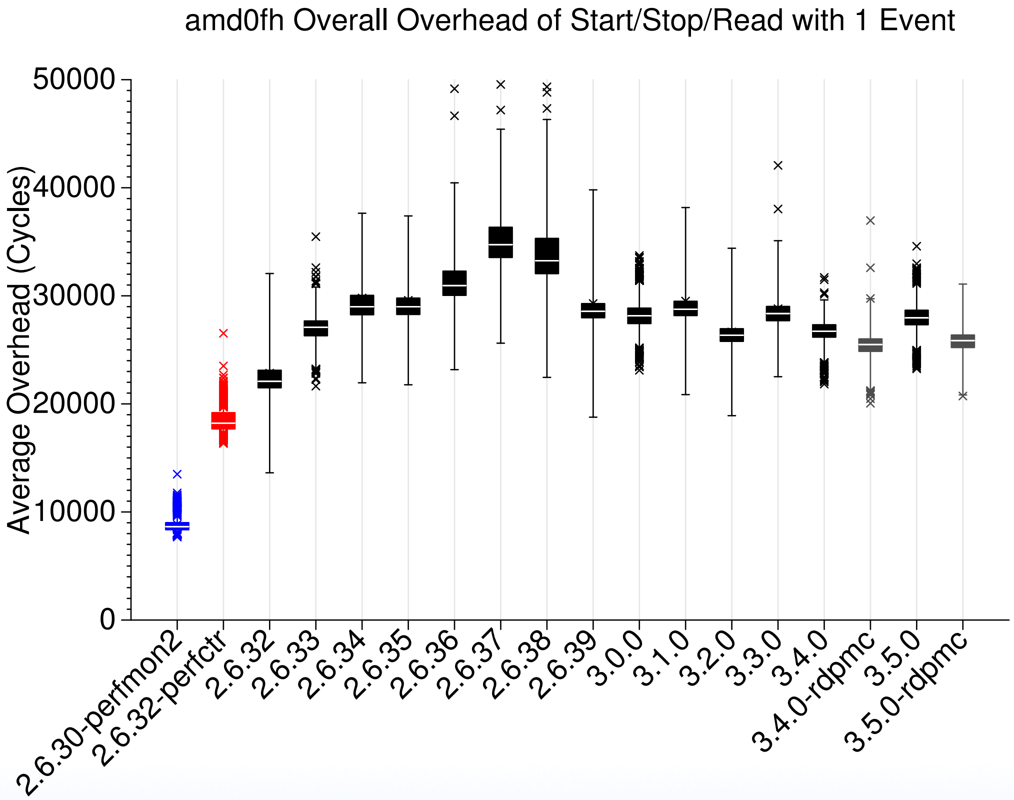
```
start_before=rdtsc();
perfctr_ioctl_w(fd,VPERFCTR_CONTROL,
                     &control, &vperfctr_control_sdesc);
start_after=rdtsc();
cstatus=kstate->cpu_state.cstatus;
nrctrs=perfctr_cstatus_nrctrs(cstatus);
retry:
   tsc0=kstate->cpu_state.tsc_start;
   rdtscl(now);
   sum.tsc = kstate->cpu_state.tsc_sum+(now-tsc0);
   for(i = nrctrs; --i >=0 ;) {
      rdpmcl(kstate->cpu_state.pmc[i].map, now);
      sum.pmc[i] = kstate->cpu_state.pmc[i].sum+
                   (now-kstate->cpu_state.pmc[i].start);
   }
   if (tsc0!=kstate->cpu_state.tsc_start) goto retry;

   read_after=rdtsc();

   _vperfctr_control(fd, &control_stop);
   stop_after=rdtsc();
```

perfmon2 Measurement Code

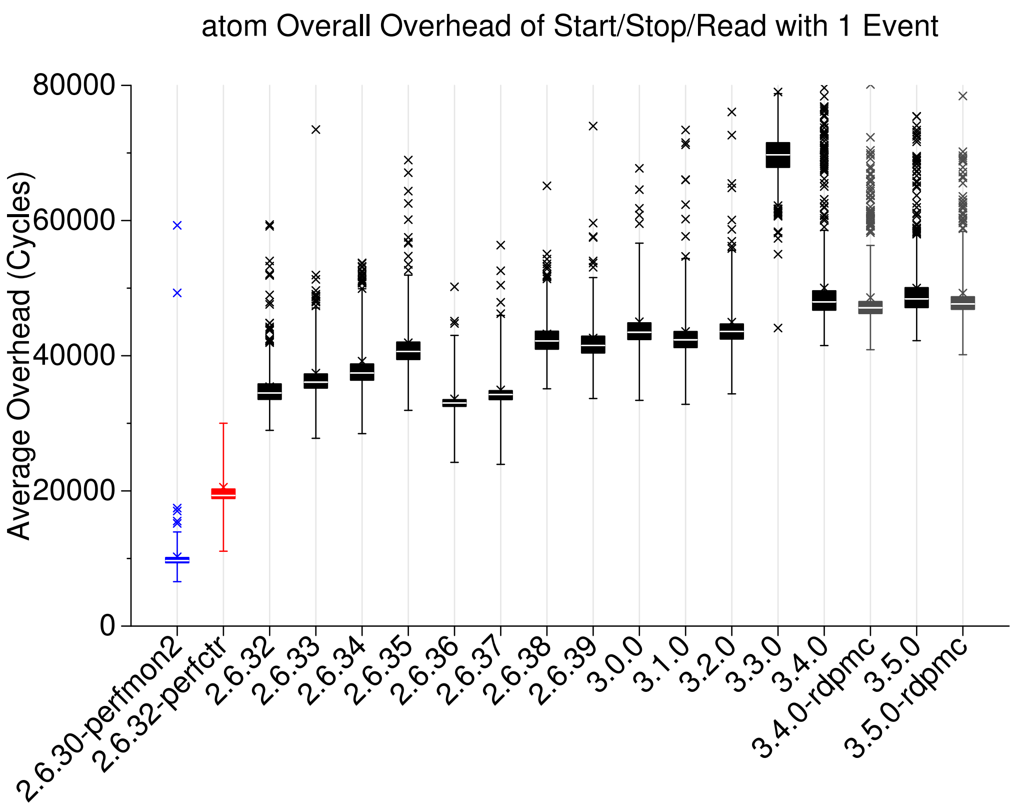
```
start_before=rdtsc();

pfm_start(ctx_fd,NULL);

start_after=rdtsc();

pfm_stop(ctx_fd);

stop_after=rdtsc();

pfm_read_pmds(ctx_fd,pd,inp.pfp_event_count);

read_after=rdtsc();
```

Overall Overhead / 1 Event, AMD Athlon64

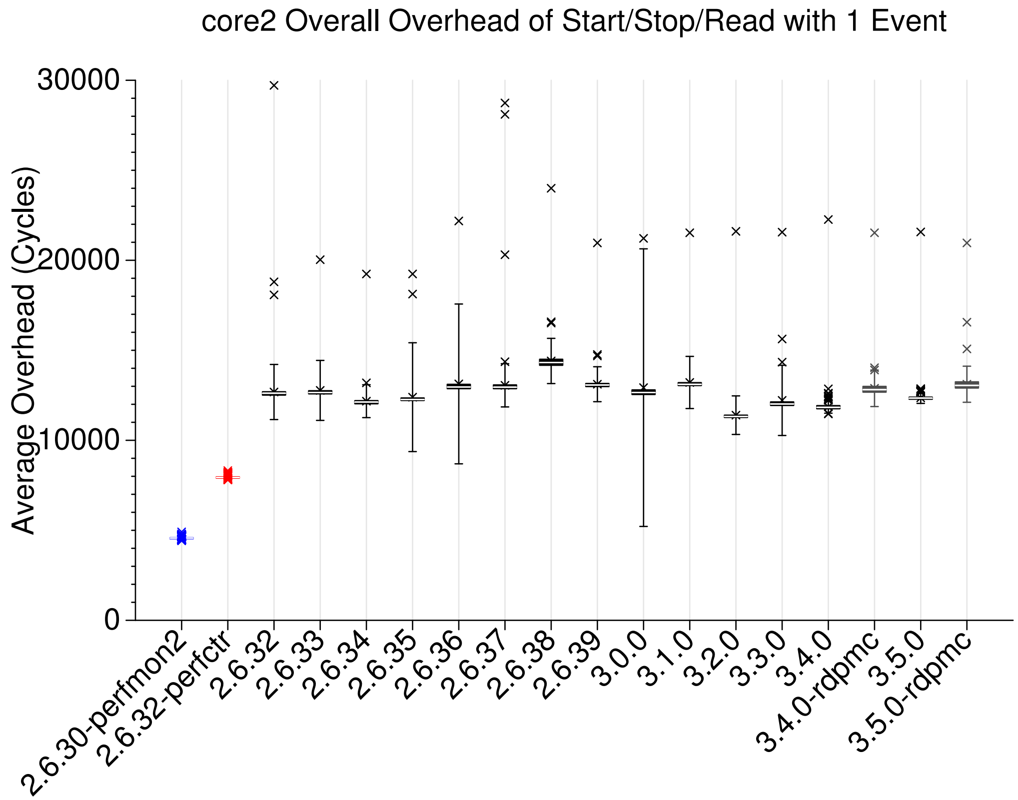
Boxplot: 25th/median/75th, stddev whiskers, outliers



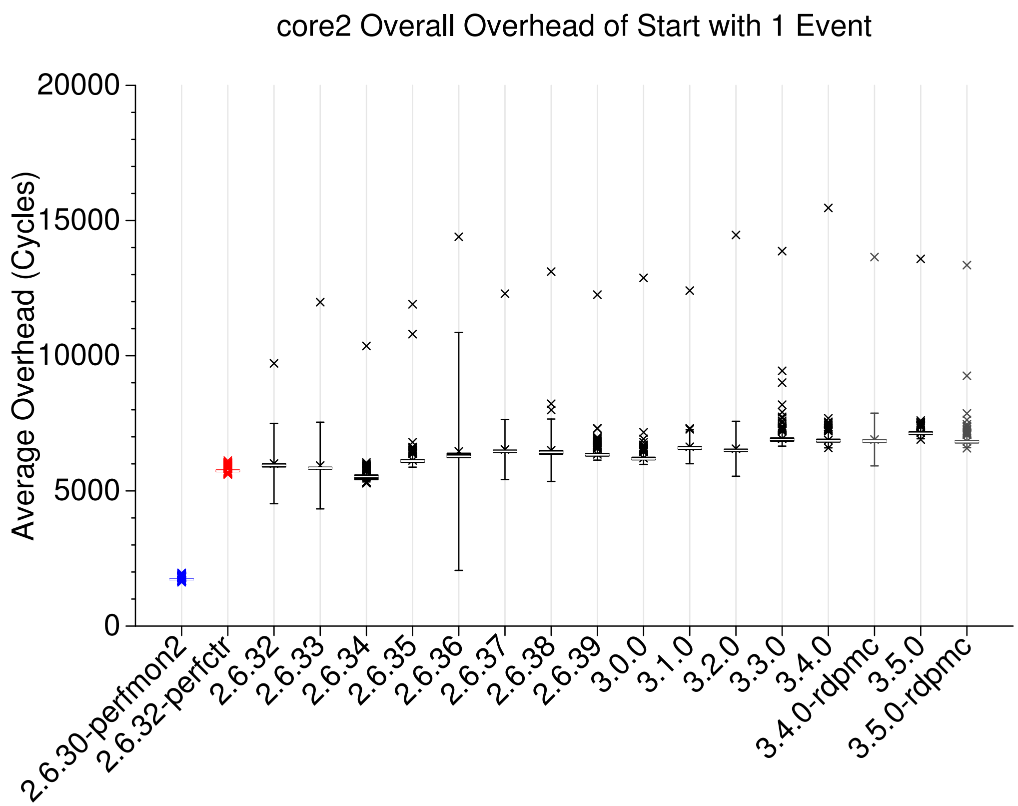
Overall Overhead / 1 Event, Intel Atom



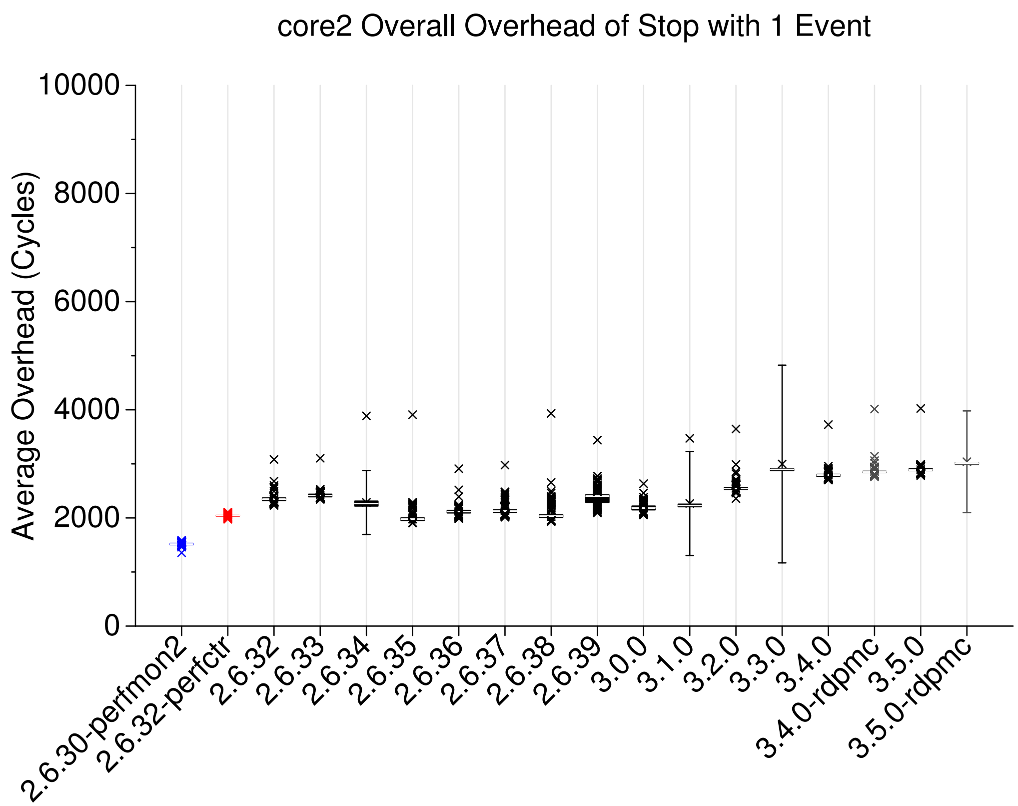
Overall Overhead / 1 Event, Intel Core2



Start Overhead / 1 Event, Intel Core2

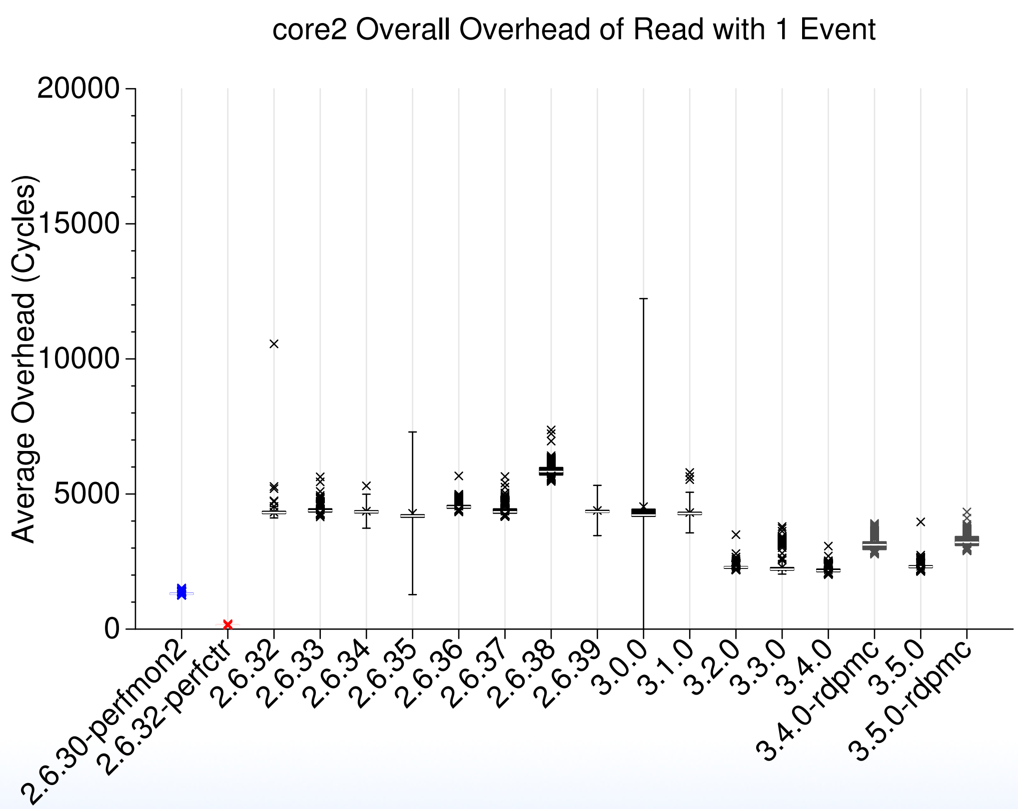


Stop Overhead / 1 Event, Intel Core2

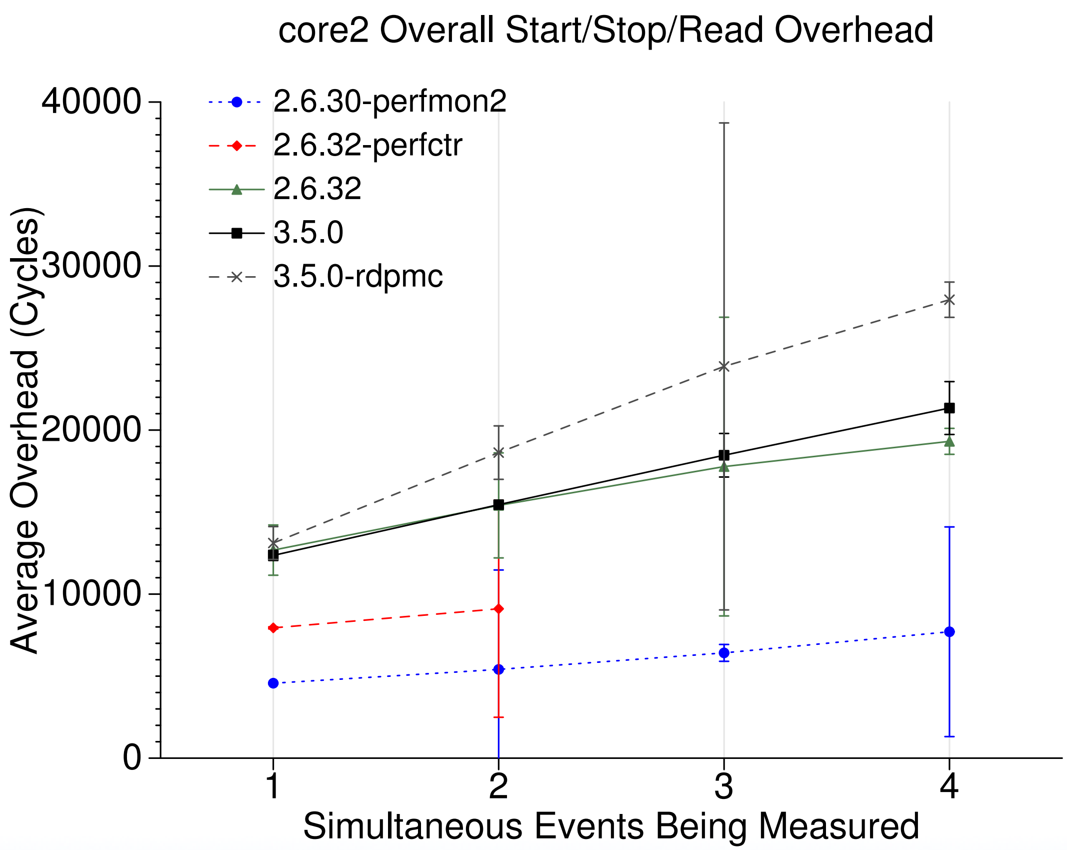


Read Overhead / 1 Event, Intel Core2

perfctr uses `rdpmc`



Overall Overhead / Multiple Events, Core2



Self-Monitoring Overhead Summary

- perfmon2 low-overhead due to very thin layer over hardware, most of work done in userspace
- perfctr has very fast `rdpmc` reads
- Some of perf_event overhead because key tasks are in-kernel and cannot be done before starting events
- Is 20,000 cycles too much to get an event count? Unclear, but perfctr is much faster, showing there is room for improvement.

New Non-perf_event Developments

- LIKWID – bypasses Linux kernel, accesses MSRs directly. Low overhead, but system-wide only, and conflicts with perf_event

- LiMiT – new patch interface similar to perfctr

Future Work

- AMD Lightweight Profiling (LWP) – (Bulldozer) events can be setup and read purely from userspace
- Intel Xeon Phi `spflt` userspace setup instruction
- Investigate causes of overhead in greater depth, as well as `rdpmc` performance issues.
- What can we learn from low overhead of perfctr and perfmon2?

Questions?

vincent.weaver@maine.edu

All code and data is available:

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