

ECE 571 – Advanced Microprocessor-Based Design Lecture 2

Vince Weaver

<https://web.eece.maine.edu/~vweaver>

vincent.weaver@maine.edu

6 September 2024

Announcements

- I've added some optional readings to the website if you want to review Computer Architecture a bit. You can access the 2007 Edition of Patterson and Hennessy for free via the UMaine Library website.
- HW#1 will be posted
- I will be handing out account username/passwords for the homework.



Account Info

- Log in to the “haswell-ep” machine for homework.
- Make sure you connect to port 2131.
- ece571-1 names are a bit impersonal.
- Use `passwd` to change your password.
- You can use `chfn` to change your name as it appears in `w` if you want.
- Please use the accounts responsibly



What do people want out of a microprocessor?

- Performance?



Review: What is Performance?

- Getting results as quickly as possible?
- Getting *correct* results as quickly as possible?
- What about Budget?
- What about Development Time?
- What about Hardware Usage?
- What about Power Consumption?
- What about Security?



Motivation for HPC Optimization

HPC environments are expensive:

- Procurement costs: \sim \$40 million
- Operational costs: \sim \$5 million/year
- Electricity costs: 1 MW / year \sim \$1 million
- Air Conditioning costs: ??



Know Your Limitation

- CPU Constrained
- Memory Constrained (Memory Wall)
- I/O Constrained
- Thermal Constrained
- Energy Constrained



Tools for Exploring Processor Performance

- Modern processors are so complicated they need to be treated as black boxes
- Does any one person fully understand?
- Can you reverse engineer by de-capping chips and looking under a microscope?
- The tools for exploring the hardware behavior overlap a lot with those used for writing optimized software



Software Tools for Performance Analysis



Simulators

- Architectural Simulators
- Can generate traces, profiles, or modeled metrics
- Slow, often 1000x or more slower
- Not real hardware, only a model
- Did I mention, slow?
- m5, gem5, simplescalar, etc



Dynamic Binary Instrumentation

- Pin, Valgrind (cachegrind), Qemu
- Still slow (10-100x slower)
- Can model things like cache behavior (can model parameters other than system running on)
- Complicated fine-tuned instrumentation can be created
- Architecture availability – Pin (no longer ARM), Valgrind, Qemu most architectures, hardest to use



Compiler (Software) Profiling

- gprof
- gcc -pg
- Adds code to each function to track time spent in each function.
- Run program, gmon.out created. Run “gprof executable” on it.
- Adds overhead, not necessarily fine-tuned, only does time based measurements.
- Pro: available wherever gcc is.



Gathering Performance Info – Aggregate Counts

- Aggregate counts (total instructions, total cycles, etc)
- Actual measurements: `perf`, `time`
- DBI measurements: `valgrind`, `qemu`
- Simulators: `gem5`, `simplescalar`



Hardware Profiling

- Insert calls on entry to function (or basic block) to track how much time spent in each
- Do you need source code?
- Manually add?
- DBI: `valgrind`
- compiler: `gprof`



Gathering Performance Info – Sampled Counts

- Sampled counts – periodically interrupt program, note the instruction pointer
- Can use info to statistically determine which part of code where most time (or other metric) is spent
- hardware: `perf`
- DBI: `valgrind`



Gathering Performance Info – Tracing

- Tracing – gather a record of every event (instruction?) that is executed. Can then replay this trace through various tools for analysis.
- Downside: huge trace files (gigabytes+)



Performance Data Analysis

Manual Analysis

- Visualization, Interactive Exploration, Statistical Analysis
- Examples: TAU, Vampir

Automatic Analysis

- Try to cope with huge amounts of data by automatic analysis
- Examples: Paradyn, KOJAK, Scalasca, Perf-expert



Evaluating Performance of Modern Systems



Benchmarks

- When measuring performance, need a reference workload to compare
- Ideally reproducible, portable, easy to compile, relevant
- Benchmarks can be gamed



Selected Commonly Seen Benchmarks

- SPEC
 - CPU 2000, CPU 2006, CPU 2017 – Commercial, Single-threaded (floating point and integer)
 - OMP – Commercial, Parallel
 - jbb – Java
- HPC Challenge – Free. HPL (Lapack). High-performance / Linear Algebra
- HPCG (conjugate gradient) new replacement for HPL
- PARSEC – Free, Multithreaded / CMP



- MiBench – Free, Embedded (2000)
- BioBench, BioParallel – Free, Bio/Data-Mining
- Imbench – Free, Operating System
- iobench – Disk I/O
- Stream – Memory



Measuring Performance



Using time

- For example

```
$ time xhpl
...
real    0m9.484s
user    0m29.150s
sys     0m7.440s
```

Real Time = Wall clock

User Time = Time used by program alone

Sys Time = Time used by OS

- When could Real be greater than User?



Other users/jobs on system.

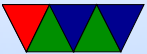
When could User be greater than Real?

Multiple threads.

- Run multiple times and notice time changes



What if Time isn't Enough?



What are Hardware Performance Counters?

- Registers on CPU that measure low-level system performance
- Available on most modern CPUs; increasingly found on GPUs, network devices, etc.
- Low overhead to read



Hardware Implementation of Counters

- Not much documentation available
- Jim Callister/Intel: “Confessions of a Performance Monitor Hardware Designer” 2005, Workshop on Hardware Performance Monitor Design
 - Transistors free, wires not. Also design time, validation, documentation, time to market. PMU has tentacles “everywhere” bringing data back to center.
 - Architect too much, lower performance, events don’t



map well to hardware. Architect too little.. software design harder.

- Which events are important? Are cache misses important if don't hurt performance? (no stalls)
- Mapping events to signal difficult. On critical path. Not enough wires. Combining signals hard if distance between wires.
- Use logging. May miss events in “shadow” of another event being logged. Use random behavior?



Learning About the Counters

- Number of counters varies from machine to machine
- Available events different for every vendor and every generation
- Available documentation not very complete (Intel Vol3b, AMD BKDG, ARM ARM/TRM)



Low-level interface

- on x86: MSRs
- ARM: CP15 system control register



Overflow

- overflows after hitting a threshold (often when wrapping, most counters are between 32 and 44 bits wide)
- One use is to keep track of counters that may wrap multiple times between reads
- If want to overflow earlier, init to a high value. So `0xc0000000` to overflow at 1 billion



Accuracy, Determinism vs Overcount

- Determinism – same count every time you run
- Overcount – an event counts more than the expected amount



SW Sources of Non-Determinism

- Accessing changing values, such as time
- Pointer-value dependencies



Linux interface

- Abstract away.
- `perf_event_open()`. See the manpage.
- Very complicated system call.
- Most people use `perf` or `PAPI` rather than calling it directly.



perf tool

A a tutorial on perf can be found here:

<https://perf.wiki.kernel.org/index.php/Tutorial>



perf tool security concerns

- You can try running on your own machine if running Linux
- For security reasons you might need to adjust `/proc/sys/kernel/perf_event_paranoid`
- You can set this to be set at boot, but haswell-ep most recent reboot didn't take.
- So let me know if somehow breaks while doing HW.



perf list

Lists available events

List of pre-defined events (to be used in `-e`):

<code>cpu-cycles OR cycles</code>	<code>[Hardware event]</code>
<code>instructions</code>	<code>[Hardware event]</code>
<code>cache-references</code>	<code>[Hardware event]</code>
<code>cache-misses</code>	<code>[Hardware event]</code>
<code>branch-instructions OR branches</code>	<code>[Hardware event]</code>
<code>branch-misses</code>	<code>[Hardware event]</code>
<code>bus-cycles</code>	<code>[Hardware event]</code>
<code>cpu-clock</code>	<code>[Software event]</code>
<code>task-clock</code>	<code>[Software event]</code>
<code>page-faults OR faults</code>	<code>[Software event]</code>
<code>minor-faults</code>	<code>[Software event]</code>
<code>major-faults</code>	<code>[Software event]</code>
<code>context-switches OR cs</code>	<code>[Software event]</code>



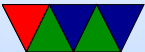
perf stat – Aggregate results

```
vince@arm:~/class/ece571$ perf stat ./matrix_multiply
Matrix multiply sum: s=27665734022509.746094
```

```
Performance counter stats for './matrix_multiply':
```

```
11585.144036 task-clock # 0.999 CPUs utilized
      19 context-switches # 0.000 M/sec
      0 CPU-migrations # 0.000 M/sec
    1,633 page-faults # 0.000 M/sec
10,343,746,076 cycles # 0.893 GHz
    5,031,717 stalled-cycles-frontend # 0.05% frontend cycles idle
    9,521,135,479 stalled-cycles-backend # 92.05% backend cycles idle
    1,176,286,814 instructions # 0.11 insns per cycle
                                # 8.09 stalled cycles per insn
    137,835,961 branches # 11.898 M/sec
      831,736 branch-misses # 0.60% of all branches

11.591796875 seconds time elapsed
```



perf stat – Specifying Events

```
vince@arm:~/class/ece571$ perf stat -e instructions,cycles ./matrix_multiply
Matrix multiply sum: s=27665734022509.746094

Performance counter stats for './matrix_multiply':

   1,174,788,622 instructions          #    0.14  insns per cycle
   8,346,588,065 cycles                #    0.000 GHz

   12.394775391 seconds time elapsed
```

Note: things like uncore events might need to specify core



perf stat – Specifying Masks

:u is user, :k kernel

ARM Cortex A9 cannot specify this distinction (results shown here are x86)

```
vince@arm:~/class/ece571$ perf stat -e instructions,instructions:u ./matri
Matrix multiply sum: s=27665734022509.746094

Performance counter stats for './matrix_multiply':

   950,526,051 instructions          #    0.00  insns per cycle
   945,661,967 instructions:u       #    0.00  insns per cycle

1.052072277 seconds time elapsed
```



libpfm4 – Finding All Event Names

```
./showevtinfo
Supported PMU models:
    [51, perf, "perf_events generic PMU"]
    [65, arm_ac8, "ARM Cortex A8"]
    [66, arm_ac9, "ARM Cortex A9"]
    [75, arm_ac15, "ARM Cortex A15"]
Detected PMU models:
    [51, perf, "perf_events generic PMU", 80 events, 1 max encoding, 0 counters, OS g
    [66, arm_ac9, "ARM Cortex A9", 57 events, 1 max encoding, 2 counters, core PMU]
Total events: 254 available, 137 supported
...
#-----
IDX      : 138412068
PMU name : arm_ac9 (ARM Cortex A9)
Name     : NEON_EXECUTED_INST
Equiv    : None
Flags    : None
Desc     : NEON instructions going through register renaming stage (approximate)
Code     : 0x74
#-----
....
```



libpfm4 – Finding Raw Event Values

```
./check_events NEON_EXECUTED_INST
Supported PMU models:
[51, perf, "perf_events generic PMU"]
[65, arm_ac8, "ARM Cortex A8"]
[66, arm_ac9, "ARM Cortex A9"]
[75, arm_ac15, "ARM Cortex A15"]
Detected PMU models:
[51, perf, "perf_events generic PMU"]
[66, arm_ac9, "ARM Cortex A9"]
Total events: 254 available, 137 supported
Requested Event: NEON_EXECUTED_INST
Actual      Event: arm_ac9::NEON_EXECUTED_INST
PMU         : ARM Cortex A9
IDX         : 138412068
Codes      : 0x74
```



perf – Using Raw Event Values

```
vince@arm:~/class/ece571$ perf stat -e r74 ./matrix_multiply  
Matrix multiply sum: s=27665734022509.746094
```

```
Performance counter stats for './matrix_multiply':
```

```
1 r74
```

```
11.303955078 seconds time elapsed
```



perf stat – multiplexing

```
perf stat -e instructions,instructions,branches,cycles,cycles ./matrix_multiply
Matrix multiply sum: s=27665734022509.746094

Performance counter stats for './matrix_multiply':

   1,178,121,057 instructions #    0.12  insns per cycle [40.23%]
   1,180,460,368 instructions #    0.12  insns per cycle [60.25%]
     138,550,072 branches                                [80.09%]
   9,999,614,616 cycles #    0.000 GHz                    [79.85%]
   9,926,949,659 cycles #    0.000 GHz                    [20.17%]

11.214630127 seconds time elapsed
```

Note same event not same results, approximate because an estimate. Percentage shown is percentage event was active during run.



perf stat – all cores

```
vince@arm:~/class/ece571$ sudo perf stat -a ./matrix_multiply
Matrix multiply sum: s=27665734022509.746094
```

```
Performance counter stats for './matrix_multiply':
```

24089.660644	task-clock	#	2.001 CPUs utilized	[100.00%]
105	context-switches	#	0.000 M/sec	[100.00%]
1,641	page-faults	#	0.000 M/sec	
9,218,451,619	cycles	#	0.383 GHz	[100.00%]
9,707,195	stalled-cycles-frontend	#	0.11% frontend cycles idle	[100.00%]
8,393,095,067	stalled-cycles-backend	#	91.05% backend cycles idle	[100.00%]
1,193,164,945	instructions	#	0.13 insns per cycle	
		#	7.03 stalled cycles per insn	[100.00%]
139,913,572	branches	#	5.808 M/sec	[100.00%]
1,221,237	branch-misses	#	0.87% of all branches	

```
12.040527344 seconds time elapsed
```

Run on *all* cores of system even if your process not there. `-a` option. Might need higher permissions. (Why?)



Security)



perf record – sampling

```
vince@arm:~/class/ece571$ time ./matrix_multiply
Matrix multiply sum: s=27665734022509.746094

real0m10.747s
user0m10.688s
sys0m0.055s
vince@arm:~/class/ece571$ time perf record ./matrix_multiply
Matrix multiply sum: s=27665734022509.746094
[ perf record: Woken up 2 times to write data ]
[ perf record: Captured and wrote 0.454 MB perf.data (~19853 samples) ]

real0m12.009s
user0m11.797s
sys0m0.203s
```

perf record creates perf.data, use -o to specify output



perf report – summary of recorded data

```
99.62% matrix_multiply matrix_multiply      [.] naive_matrix_multiply
 0.38% matrix_multiply [kernel.kallsyms].head.text [k] 0xc0046a54
 0.00% matrix_multiply ld-2.13.so          [.] _dl_relocate_object
 0.00% matrix_multiply [kernel.kallsyms]          [k] __do_softirq
```

Our benchmark is simple (only one function) so the profiled results are not that exciting.

The [k] indicates that profile happened while the kernel was running.



Similar ways to get Similar Results

- Valgrind/Callgrind

valgrind - -tool=callgrind BENCHMARK

then run *callgrind_annotate*

Note Valgrind is probably around 50 times slower

- Use gprof

Compile your code with *-pg*

Run *gprof BENCHMARK*



perf annotate – show hotspots in assembly

```
0.00 :      845a:      vldr    d7, [pc, #124] ; 84d8 <naive_matrix_m
30.97 :      845e:      adds   r1, r4, r3
1.43 :      8460:      add.w  r3, r3, #4096 ; 0x1000
1.17 :      8464:      adds   r2, #8
1.36 :      8466:      cmp.w  r3, #2097152 ; 0x200000
2.97 :      846a:      vldr   d5, [r2]
2.62 :      846e:      vldr   d6, [r1]
2.78 :      8472:      mov    r9, r2
2.42 :      8474:      vmla.f64      d7, d5, d6
53.81 :      8478:      bne.n  845e <naive_matrix_multiply+0x72>
0.01 :      847a:      adds   r5, #1
```

Note: recent perf default to cycles:p The annotated results show a branch and an add instruction accounting for 83% of profiles. Likely this is due to skid and the key instruction is the previous `vmla.f64` floating point multiply instruction. The processor just isn't able to stop at the exact instruction when the interrupt comes



in.



Skid

- Beware of “skid” in sampled results
- This is what happens when a complex processor cannot stop immediately, so the reported instruction might be off by a few instructions.
- Some processors do not have this problem, other Intel processors have special events that can compensate for this.

