

LECTURE 18

HARDWARE PERFORMANCE COUNTERS

The simplest hardware-aided performance-measuring tool is:
the **time stamp counter** (TSC)

- Introduced by **Intel** with the Pentium architecture (1993)
- Similar feature available on **ARM** since ARMv7 (1996)
- Special integer register
- Incremented by one at a constant rate (e.g. every clock cycle)
- Reading this register has high latency (>10 cycles)
- Useful for microbenchmarks and instrumentation
- `time.time()` / `clock_gettime()` use this internally

More complex performance counters

Since then, [Intel](#) and [ARM](#) have added many more performance counters:

- executed (“retired”) instructions
- branches
 - successfully predicted
 - mispredicted branches
- memory accesses
 - found in L1 cache
 - L1 misses, found in L2 cache
 - L2 misses, found in (last-level) L3 cache
 - L3 misses, found in main memory
 - TLB (page table cache) hits
 - TLB misses
- **Pros**
 - always measured
 - no performance penalty
 - no interference with normal execution
- **Cons**
 - only an aggregate measure (totals)

Linux perf

```
perf stat ./application
```

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

3,216.90	msec	task-clock	#	1.000	CPUs utilized	
8		context-switches	#	2.487	/sec	
1		cpu-migrations	#	0.311	/sec	
6,205		page-faults	#	1.929	K/sec	
9,442,508,623		cycles	#	2.935	GHz	(52.90%)
7,596,331,032		instructions	#	0.80	insn per cycle	(58.81%)
1,086,117,213		branches	#	337.629	M/sec	(58.84%)
1,085,287		branch-misses	#	0.10%	of all branches	(58.87%)
2,162,685,901		L1-dcache-loads	#	672.289	M/sec	(58.87%)
1,079,393,101		L1-dcache-load-misses	#	49.91%	of all L1-dcache accesses	(58.88%)
1,069,062,732		LLC-loads	#	332.327	M/sec	(58.87%)
6,537,301		LLC-load-misses	#	0.61%	of all L1-icache accesses	(23.50%)
2,161,850,109		dTLB-loads	#	672.029	M/sec	(23.50%)
896,301		dTLB-load-misses	#	0.04%	of all dTLB cache accesses	(23.50%)
9,051,173		dTLB-stores	#	2.814	M/sec	(23.50%)
81,624		dTLB-store-misses	#	25.374	K/sec	(23.50%)

```
3.217829387 seconds time elapsed
```

```
3.167788000 seconds user
```

```
0.022723000 seconds sys
```

STOCHASTIC INSTRUMENTATION

Limitations of performance counters

- How could we find **hot spots**
(small groups of instructions that the application spends a lot of time running)
- What about performance counts (cache misses, mispredicted branches,...)
at those **hot spots**?
- Instrumentation is expensive (and affects accuracy)

Solution: stochastic instrumentation

- every N cycles (e.g. every 1,000,000th cycle / every 0.1ms), a **sample** is taken
- the **sample** records:
 - which instruction is currently being executed
 - optionally, what it is waiting for (instr. decoding, pipeline bubble, memory access, ...)
 - optionally, instruction addresses of the last few branches
 - optionally, whether those branches were successfully predicted

Stochastic instrumentation

- Pros
 - no performance penalty
 - no interference with normal execution
 - accuracy naturally increases on hotspots
- Cons
 - none

Analysis applications

- Linux
 - `perf record / perf report`
 - KDAB hotspot
- MacOS: Apple XCode Instruments
- Windows: Visual Studio (“dynamic instrumentation” / “collection via sampling”)
- Intel-specific: vTune
- AMD-specific: uProf

Bottom-up analysis

Intel VTune Profiler

Welcome x r018ue x

Microarchitecture Exploration

Analysis Configuration Collection Log Summary **Bottom-up** Event Count Platform

Grouping: Function / Call Stack

Function / Call Stack	CPU Time	Clockticks	Instructions Retired	CPI Rate	Retiring	Front-End Bound
p_1gs_Ax_1	223.864ms	300,000,000	1,198,800,000	0.250	32.5%	22.9%
p_2gs_Ax_2	214.773ms	270,000,000	2,712,000,000	0.100	76.8%	6.5%
p_spx_dual_violated	193.182ms	254,400,000	1,929,600,000	0.132	43.5%	12.6%
func@0x92e0	87.500ms	264,000,000	261,600,000	1.009	41.3%	4.1%
p_spx_dse_update_w	84.659ms	112,800,000	1,429,200,000	0.079	73.2%	1.0%
p_1gs_Ax_2	84.091ms	108,000,000	906,000,000	0.119	89.9%	10.9%
p_bfrt_el	81.818ms	112,800,000	528,000,000	0.214	24.1%	12.1%
p_vec2_fmsub	80.682ms	134,400,000	1,194,000,000	0.113	31.2%	7.4%
p_spx_dual_full_leaving	70.455ms	133,200,000	100,800,000	1.321	37.2%	8.7%
func@0xa3a0	51.705ms	168,000,000	362,400,000	0.464	76.6%	21.9%
p_spx_compute_row_sib_dai	51.705ms	81,600,000	298,800,000	0.273	23.3%	7.8%
p_bfrt_select	49.432ms	61,200,000	408,000,000	0.150	48.6%	11.3%
p_map_qlookup	42.045ms	46,800,000	61,200,000	0.765	8.3%	1.4%
p_spx_dse_update_xB	40.909ms	57,600,000	457,200,000	0.126	45.4%	8.1%
p_fold_iter	33.523ms	0	162,000,000	0.000	15.1%	1.9%
p_vec_z_scatter	29.545ms	50,400,000	228,000,000	0.221	31.1%	3.9%

Microarchitecture Usage: 32.5% of Pipeline Slots

μPipe

- Retiring: 32.5% of Pipeline Slots
- Front-End Bound: 22.9% of Pipeline Slots
- Front-End Latency: 14.6% of Pipeline Slots
- ICache Misses: 0.0% of Clockticks
- ITLB Overhead: 0.0% of Clockticks

Thread

- lpopt (TID: 369669)
- gzip (TID: 369688)
- amplx-runss (TID: 369669)
- lpopt (TID: 369688)

CPU Time

FILTER 100.0% Any Process Thread Any Thread Module Any Module Call Stack Mode User functions Loop Mode Functions only Inline Mode Show inline fun

