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

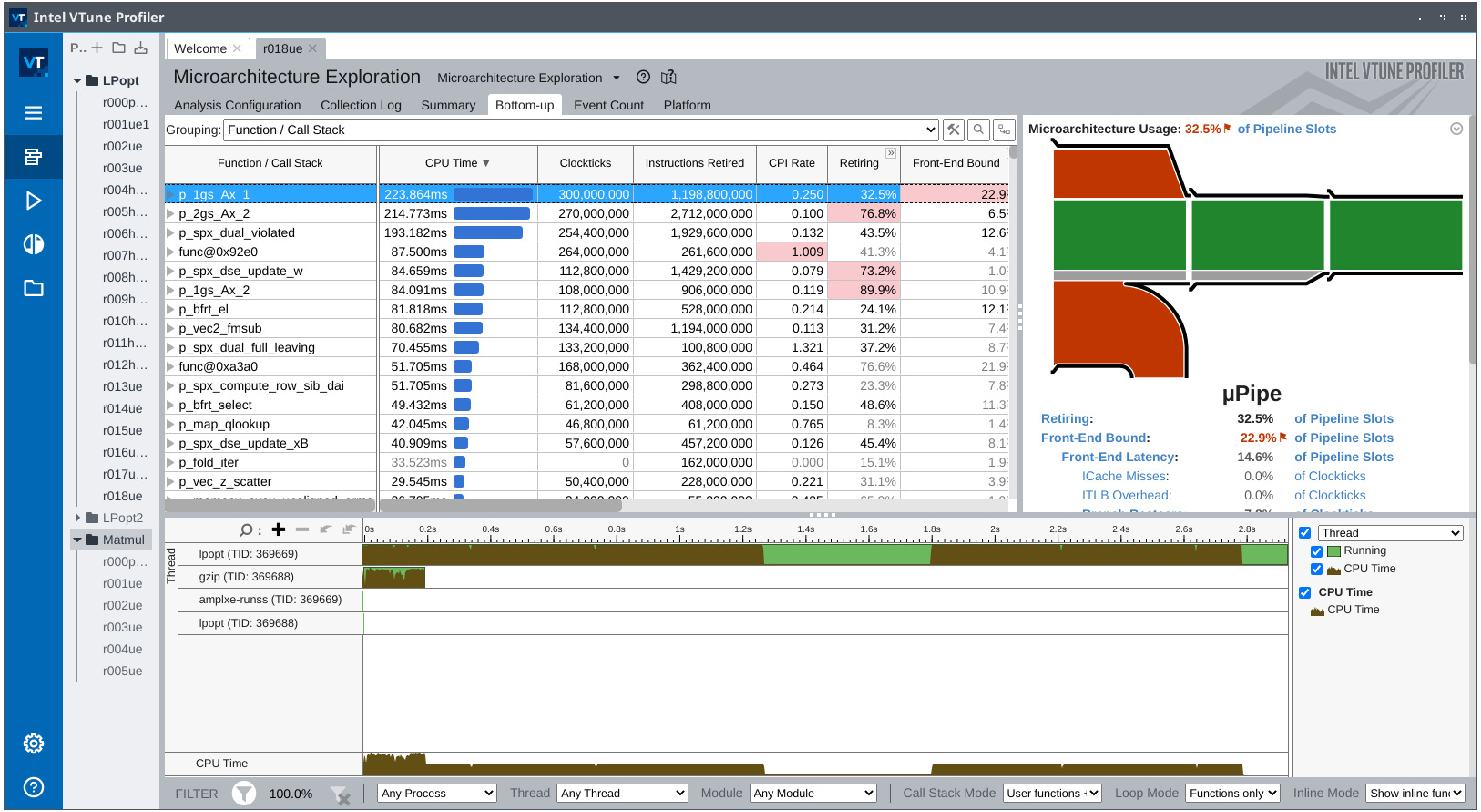
- Pros
  - no performance penalty
  - no interference with normal execution
  - accuracy naturally increases on hotspots

- 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



Flame graphs