ECE 471 – Embedded Systems
Lecture 33

Vince Weaver

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

vincent.weaver@maine.edu

3 December 2018

Announcements

- N/A

Introduction to Performance Analysis

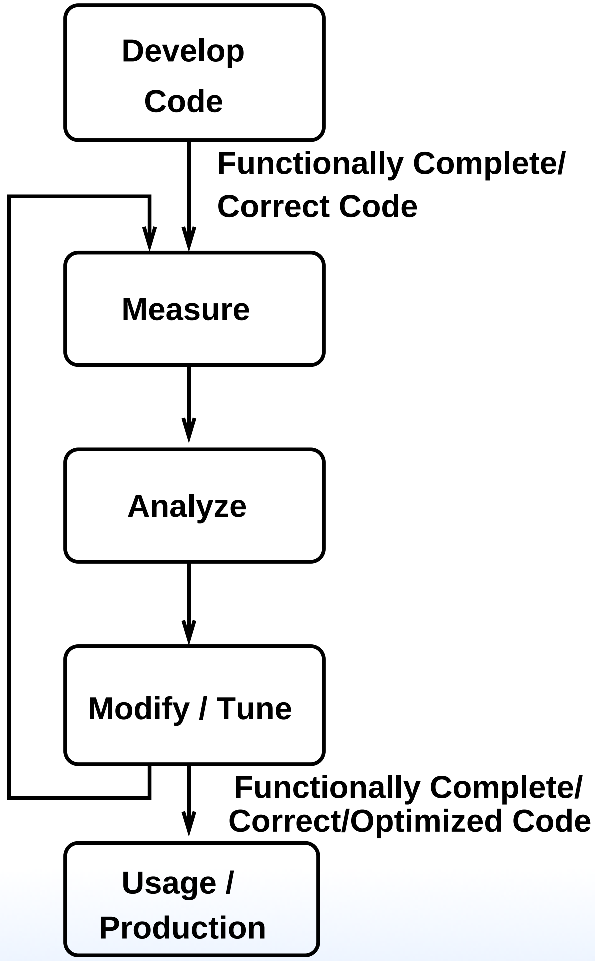
What is Performance?

- 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?

Know Your Limitation

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

Performance Optimization Cycle



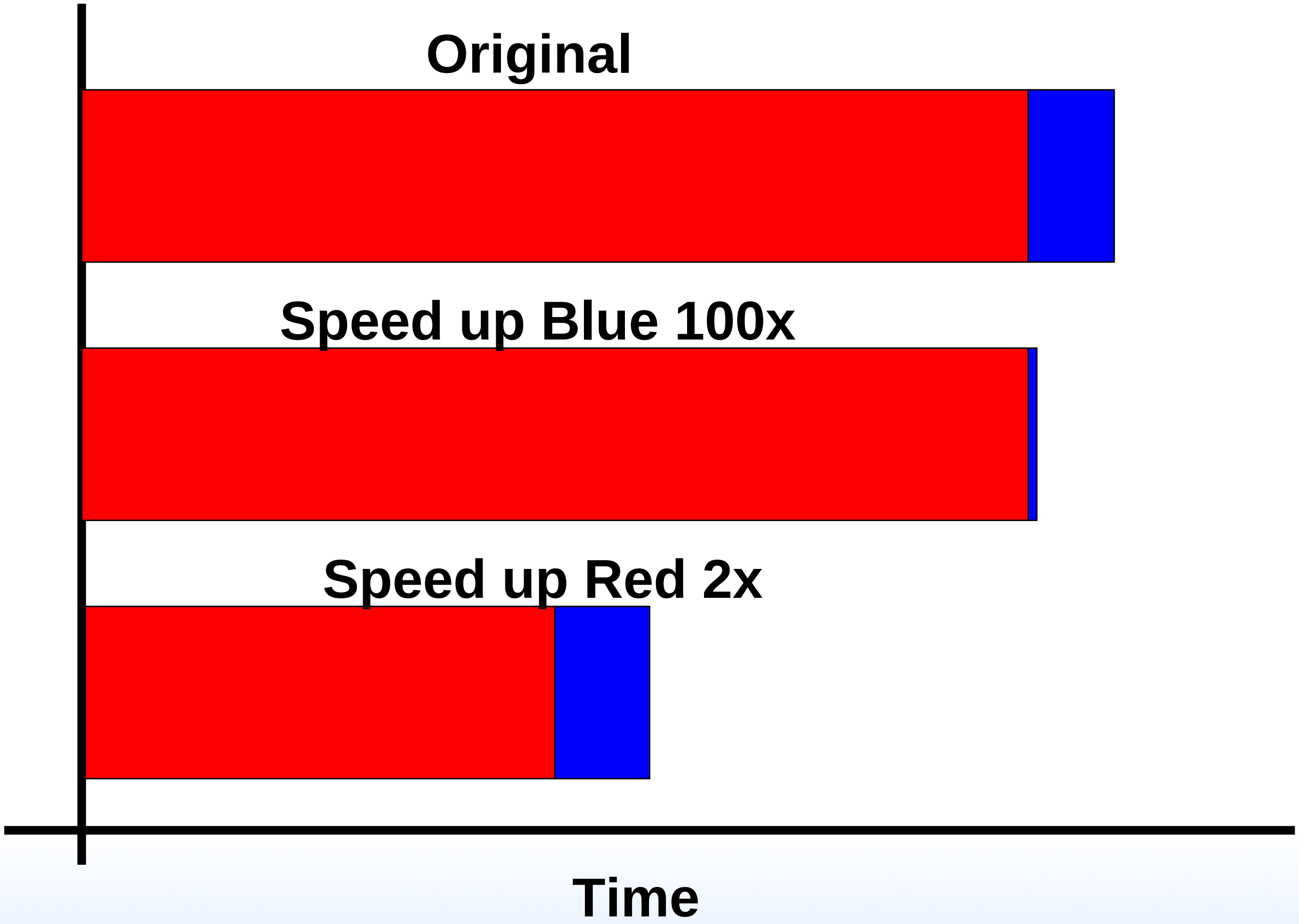
Wisdom from Knuth

"We should forget about small efficiencies, say about 97% of the time:

premature optimization is the root of all evil.

Yet we should not pass up our opportunities in that critical 3%. A good programmer will not be lulled into complacency by such reasoning, he will be wise to look carefully at the critical code; but only after that code has been identified" — Donald Knuth

Amdahl's Law



Measuring Time

- Already talked about Power, but other aspect is speed (time)

- `time` command

- Reports real (wall-clock), user (used by program), sys (kernel)

- In virtualized systems wall-clock time might become meaningless

- Timers, rdtsc?
- When can user time exceed real? (multi-threaded)
- When can user+sys be less than real? (If something else is using the system)
- Waiting on I/O and Interrupts count as sys time.

Using "time"

```
vince@rasp-pi5 ~/research/libpfm4/examples $ time
check_events check_events.o showevtinfo    showev
check_events.c Makefile showevtinfo.c

real 0m0.018s
user 0m0.010s
sys 0m0.000s
```

What do they mean? Can real be higher than user? Can user be more than real? Is it deterministic (will it vary run to run)

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

Low-level interface

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

CP15 registers in Pi

- BCM2835 (Original Pi)
  - 3 counters available (1 cycle counter, 2 generic)
  - 25 events
  - No way to specify kernel vs user
  - On Raspberry Pi original overflow interrupt not connected
- BCM2836 (Pi2)
  - The ARM-Cortex A7 has 5 counters
  - Can specify kernel, user

- Overflow works

- BCM2837 (Pi3)
  - The ARM-Cortex A53 has 7 counters
  - Can specify kernel, user
  - Overflow works

CP15 Interface

- use `mcr`, `mrc` to move values in/out

  ```
  MRC p15,0,Rt,c9,c12,0
  MCR p15,0,Rt,c9,c12,0
  ```

- Two EVNTCNT registers
- Cycle Counter register
- Two Event Config registers
- Count enable set/clear, count interrupt enable/clear,

overflow, software increment

- PMU management registers
- in general only privileged access (why) but can be configured to let users access.

Hardware Performance Counters: The Operating System Interface

Operating System Interface

A typical operating system performance counter interface will provide the following:

- A way to select which events are being monitored
- A way to start and stop counting
- A method of reading counter results when finished, and
- If the CPU supports notification on counter overflow, some mechanism for passing on overflow information

Operating System Interface

Some operating systems provide additional features:

- Event scheduling: often there are limitations on which events can go into which counters,
- Multiplexing: the OS can hide the fact that only a limited number of counters are available by swapping events in and out and extrapolating counts using time accounting,
- Per-thread counting: by loading and saving counter

values at context switch time a count specific to a process can be achieved,

- Attaching to a process: counts can be taken from an already running process, and

- Per-cpu counting: as with per-thread counting, counts can be accumulated per-cpu.

Older Linux Interfaces

- Historical – typically just exported msrs
- Oprofile – only does profiling
- Perfctr – good but required kernel patch
- Perfmon2 – was making headway until perf_event came from nowhere and became official

perf_event

- Developed from scratch in 2.6.31 by Molnar and Gleixner
- Everything in the kernel
- `perf_event_open()` syscall (manpage still under development)
- perf_event_attr structure with 40 complex interdependent parameters
- ioctl() system call to enable/disable

- read() system call to read values
- can gather sampled data in circular buffer
- can get signal on overflow or full buffer

perf_event Generalized Events

- perf_event provides support for “common” generalized events

- makes things easier for user at expense of papering over the differences between events

- events need to be validated to make sure they are providing useful results

perf_event Generalized Events Issues

- Which event to choose (Nehalem)
- From 2.6.31 to 2.6.35 AMD "branches" was taken not total
- Nehalem L1 DCACHE reads.
  PAPI uses `L1D_CACHE_LD:MESI`;
  perf uses `MEM_INST_RETIRED:LOADS`

perf_event Event Scheduling

- Some events have hardware constraints. Can only be in one counter
- You can do this scheduling in userspace; lets the algorithm be changed more easily
- Scheduling can be expensive; do so at event start can slow things down.

perf_event Multiplexing

- You may wish to measure more events simultaneously than hardware can support (NMI watchdog may steal one too)
- perf_event supports this in-kernel (you can also do this in userspace)
- there are various ways to try to ensure good statistical results. in kernel you have to trust the kernel programmers.

perf_event Event Names

- Event names are provided in the hardware manuals, but can be inconsistent
- Traditionally used libraries to provide names. libpfm4
- perf tool is starting to provide own list of events (they refuse to link libpfm4) that are based on a hybrid of libpfm4 and kernel names
- Also some event names are provided by the kernel under `/sys`

perf_event Software Events

- perf_event provides internal kernel events through same interface
- `page-fault`, `task-clock`, `cpu-clock`, etc.

perf_event Perf Tool

- Included with kernel source code
- Tied to kernel, but backwards compatible
- Most kernel devs use this rather than outside tools
- apt-get install linux-perf (new) or linux-tools (old)

perf

Based on a tutorial found here:
https://perf.wiki.kernel.org/index.php/Tutorial

perf list

```
Lists available events
List of pre-defined events (to be used in -e):
  cpu-cycles OR cycles                              [Hardware event]
  instructions                                      [Hardware event]
  cache-references                                  [Hardware event]
  cache-misses                                      [Hardware event]
  branch-instructions OR branches                   [Hardware event]
  branch-misses                                     [Hardware event]
  bus-cycles                                        [Hardware event]

  cpu-clock                                         [Software event]
  task-clock                                        [Software event]
  page-faults OR faults                             [Software event]
  minor-faults                                      [Software event]
  major-faults                                      [Software event]
  context-switches OR cs                            [Software event]
```

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_multip
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
```

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 running there. -a option. Need root permissions

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.

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
```

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.