ECE 571 – Advanced Microprocessor-Based Design Lecture 7

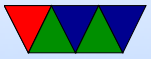
Vince Weaver

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

vincent.weaver@maine.edu

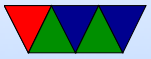
5 February 2013

Hardware Performance Counters – Software Tools



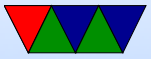
First Some Assembly Optimization Review

Just to follow up on Homework 1.



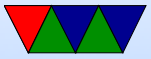
Print Number Explanations

- Code was optimized x86 code direct-ported to ARM. This explains some of the design decisions.
- Why 11-byte buffer in BSS?
  Max size of a 32-bit integer is roughly 4 billion, fills 10 digits. The NUL-termination is set to 0 by free by BSS (11th byte)
- Why go backward?
  That's the easiest way to convert to decimal, when dividing by 10. Also it makes leading zero removal



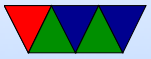
easier.

- Leading zero removal but still print 0 if only 0. Go through and print once, even if zero.
- Why use fancy addressing to subtracting always, but then have to adjust by -1 at end?
  This made more sense in the original x86 code where the autodecrement instruction was more compact that way.



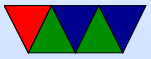
Other Common Optimization Tricks

- xor with self – at least on x86 fastest way to set to 0
- multiply by power of 2 – just shift left (though on some HW implementations add with self faster than single left shift)
- multiply by constant – just shifts and adds
- div/mod of power of 2 – divide just shift right. remainder is just and with mask.



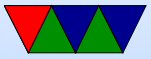
Ways to Divide by 10

- hardware div instruction
- BCD, shift by 8 (most machines don't have BCD in hardware)
- 32x32 with 64-bit result multiply reciprocal
- multiply by fraction
- lots of shifts (useful if no multiply instruction)
- iterative subtraction (useful if no mul either)



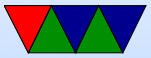
Simple Timing Analysis of Divide

- using `time` command.
  Real time is wallclock,
  User is how much actual code used (ignores other processes on system),
  Sys is how much kernel used



arm div by 10 analysis

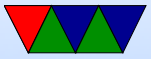
- div – (not available on most ARM chips)
- high-mul – fastest, but `umull` instruction not available on THUMB
- shifts – slower than high-mul, (needed on armv4 with no `umull` instruction)



C Compiler – 7 million div/10

Simple C benchmark does 7 million divides with remainder.

Much of overhead comes from printing? Why must I print? Simple way to keep the compiler from optimizing away the divide (if I don't output the result, then it doesn't need to be executed)

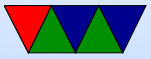


C Compiler – arm

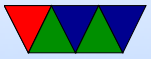
gcc -O0	`umull 0xcccccccd, >>3, again, shift`	10.02s
gcc -O2	`umull 0xcccccccd, >>3, shift`	9.90s
gcc -Os	`__aeabi_uidivmod`	11.40s

calls __aeabi_uidiv and subtracts, shift/subtract

15% speedup

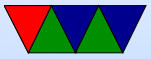


Performance Analysis



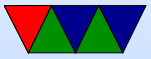
Benchmarks

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

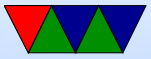


Selected Commonly Seen Benchmarks

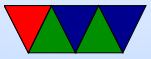
- SPEC
  - CPU 2000, CPU 2006 – Commercial, Single-threaded CPU benchmarks (floating point and integer)
  - OMP – Commercial, Parallel
  - jbb – Java
- HPC Challenge – Free. HPL (Linpack). High-performance / Linear Algebra
- PARSEC – Free, Multithreaded / CMP



- MiBench – Free, Embedded (2000)
- BioBench, BioParallel – Free, Bio/Data-Mining
- lmbench – Free, Operating System

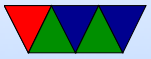


Higher Level Tools



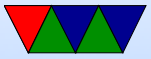
perf

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



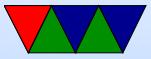
perf list

```
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]
```



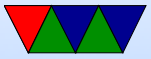
perf stat – Aggregate results

```
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
```

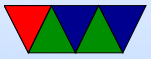


perf stat – Specifying Masks

:u is user, :k kernel

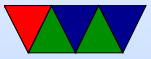
ARM Cortex A9 cannot specify this distinction (results shown here are x86)

```
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
```



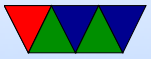
libpfm4 – Finding All Event Names

```
./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
#----------------------------
....
```



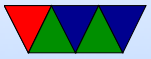
libpfm4 – Finding Raw Event Values

```
./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

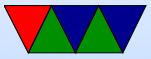
```
vince@arm:~/class/ece571$ perf stat -e r74 ./matrix_multiply
Matrix multiply sum: s=27665734022509.746094

 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
```

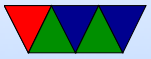
Note same event not same results, approximate because an estimate. Percentage shown is percentage event was active during run.



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
```

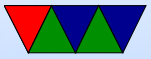
Run on *all* cores of system even if your process not running there. -a option. Need root permissions



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

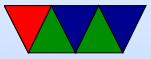


perf report – summary of recorded data

```
 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.

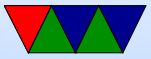
The `[k]` indicates that profile happened while the kernel was running.



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.



Homework Will Be Posted This Afternoon