ECE 471 – Embedded Systems
Lecture 24

Vince Weaver

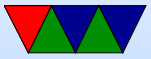
http://www.eece.maine.edu/~vweaver

vincent.weaver@maine.edu

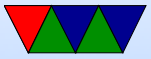
3 December 2013

Announcements

- Project – remember status update is due today
- HW#5 – has been assigned, due next Tuesday

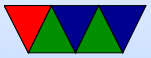


Introduction to Performance Analysis



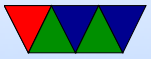
What is Performance?

- 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?

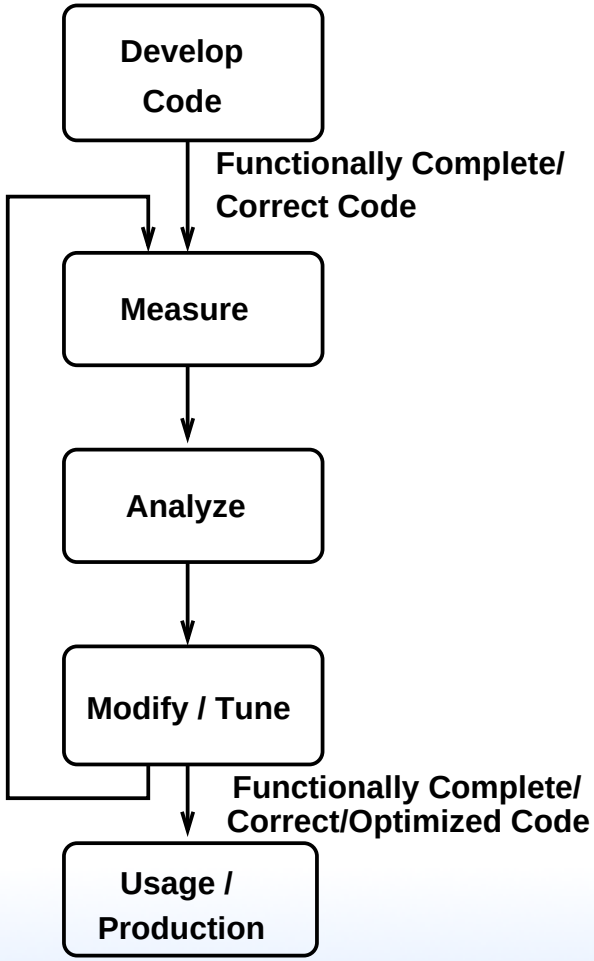


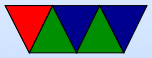
Know Your Limitation

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



Performance Optimization Cycle



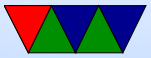


Wisdom from Knuth

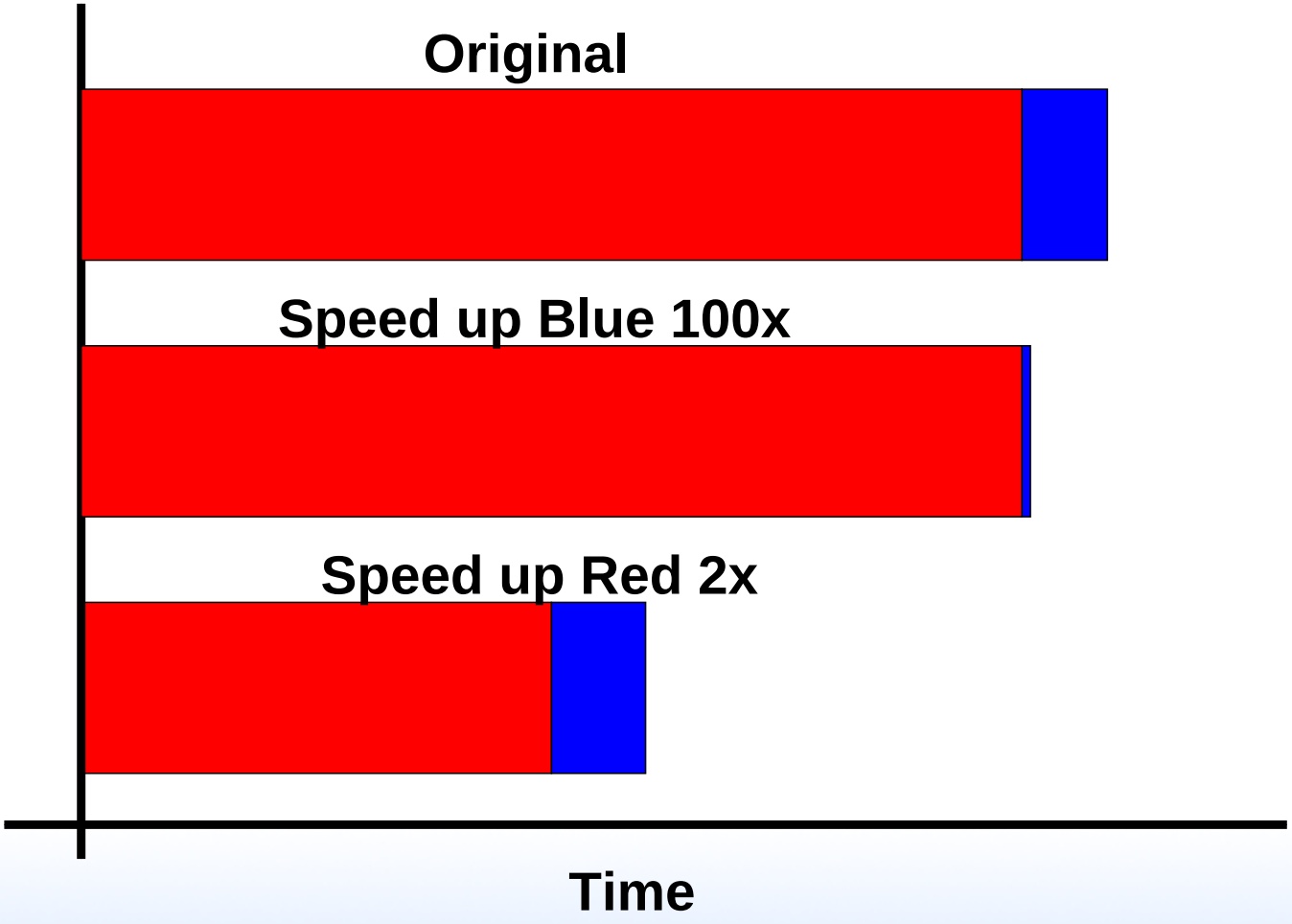
"We should forget about small efficiencies, say about 97% of the time:

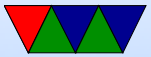
premature optimization is the root of all evil.

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



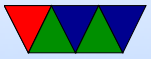
Amdahl's Law



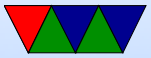


Measuring Time

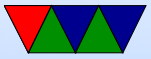
- Already talked about Power, but other aspect is speed (time)
- `time` command
- Reports real (wall-clock), user (used by program), sys (kernel)
- In virtualized systems wall-clock time might become meaningless



- Timers, rdtsc?
- When can user time excede 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.

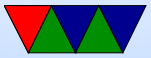


Software Tools for Performance Analysis



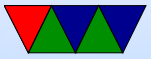
Simulators

- 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



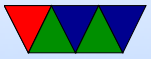
Dynamic Binary Instrumentation

- 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



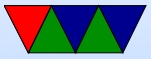
Compiler Profiling

- 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

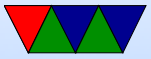


time based measurements.

- Pro: available wherever gcc is.

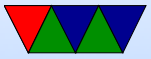


Hardware Tools for Measuring Performance



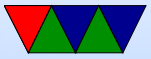
What are Hardware Performance Counters?

- 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



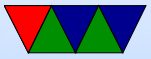
Hardware Implementation of Counters

- 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



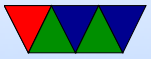
map well to hardware. Architect too little.. software design harder.

- 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?



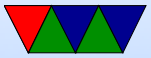
Learning About the Counters

- 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)



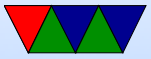
Low-level interface

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



CP15 registers on Cortex A9

- 6 counters available
- 58 events, 17 architectural, 41 A9 Specific, split between Approximate, Precise
- No way to specify kernel vs user (Cortex A15 does?)
- Cortex A9 has bug where PMU interrupts may be lost

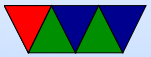


CP15 Interface

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

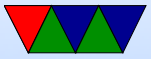
  ```
  MRC p15,0,Rt,c9,c12,0
  MCR p15,0,Rt,c9,c12,0
  ```

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

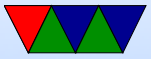


overflow, software increment

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

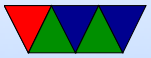


Hardware Performance Counters: The Operating System Interface



Operating Systems

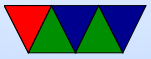
- UNIX – long history of support
- Windows – no native support (can get Intel Vtune)
- OSX – no native support (can get shark)
- Linux – On 95% of Top 500 computers, many embedded systems



Operating System Interface

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

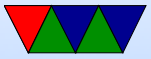
- A way to select which events are being monitored
- A way to start and stop counting
- A method of reading counter results when finished, and
- If the CPU supports notification on counter overflow, some mechanism for passing on overflow information



Operating System Interface

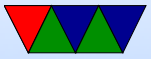
Some operating systems provide additional features:

- 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



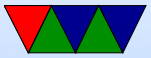
values at context switch time a count specific to a process can be achieved,

- 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.



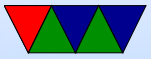
Older Linux Interfaces

- 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

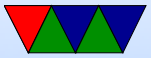


perf_event

- 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

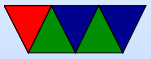


- read() system call to read values
- can gather sampled data in circular buffer
- can get signal on overflow or full buffer



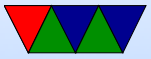
perf_event Generalized Events

- 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



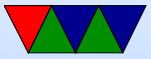
perf_event Generalized Events Issues

- 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`



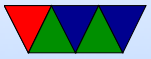
perf_event Event Scheduling

- 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.



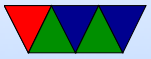
perf_event Multiplexing

- 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.



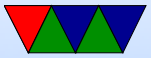
perf_event Event Names

- 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`



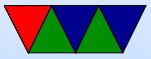
perf_event Software Events

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



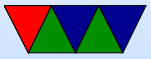
perf_event Perf Tool

- Included with kernel source code
- Tied to kernel, but backwards compatible
- Most kernel devs use this rather than outside tools



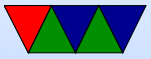
rdpmc instruction

- Allow users direct reads of performance counters w/o system call



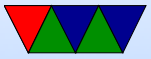
non-CPU counters

- things like network cards, GPUs, etc.



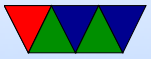
perf

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



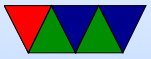
perf list

```
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]
```



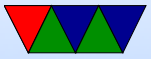
perf stat – Aggregate results

```
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
```

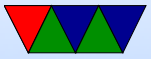


perf stat – Specifying Masks

:u is user, :k kernel

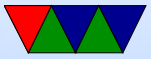
ARM Cortex A9 cannot specify this distinction (results shown here are x86)

```
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
```



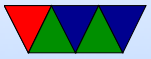
libpfm4 – Finding All Event Names

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



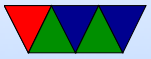
libpfm4 – Finding Raw Event Values

```
./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

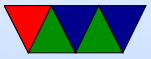
```
vince@arm:~/class/ece571$ perf stat -e r74 ./matrix_multiply
Matrix multiply sum: s=27665734022509.746094

 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
```

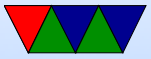
Note same event not same results, approximate because an estimate. Percentage shown is percentage event was active during run.



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
```

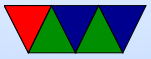
Run on *all* cores of system even if your process not running there. -a option. Need root permissions



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

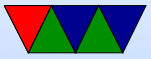


perf report – summary of recorded data

```
 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.

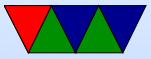
The `[k]` indicates that profile happened while the kernel was running.



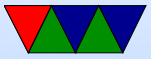
perf annotate – show hotspots in assembly

```
 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.

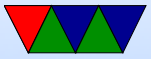


Hardware Performance Counters – Software Tools

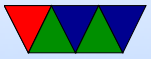


PAPI (Performance API)

- Low-level Performance Measurement Interface
- Cross-platform
- Self-monitoring or Sampling
- C, C++, Fortran (or attach to running process)
- Basis for more advanced visualization tools. Vampir, Tau, PerfExpert, etc.

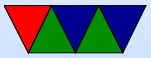


- Provides high-level access to timers
- Provides high and low-level access to performance counters
- Provides profiling support
- Provides system information
- Components
- Fine-grained instrumentation



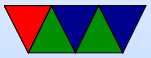
PAPI Limitations

- In general have to modify source code
- Overhead included in program run



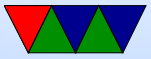
PAPI Platforms

- Linux perf_event
- Linux perfmon2/perfctr (mostly deprecated except Cray)
- IBM BlueGene P/Q
- Solaris
- FreeBSD
- IBM AIX



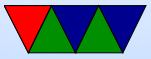
PAPI CPUs

- x86, MIC
- ARM
- Power
- SPARC
- Itanium
- MIPS

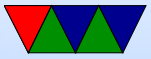


PAPI Components

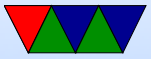
- Appio – I/O bandwidth
- BGPM – IBM Bluegene extra
- Coretemp – chip temp sensors, etc.
- CUDA – NVidia GPU
- Infiniband – high-speed network
- lmsensors – chip sensors



- lustre – parallel filesystem
- micpower – power on Intel MIC (Xeon PHI)
- MX – myrinet, high-speed network
- net – generic Linux network
- NVML – Nvidia power
- RAPL – Intel Sandybridge/Ivybridge Power
- Stealtime – Virtual Machine stealtime



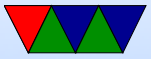
- VMware – VMware stats



PAPI Tools

Note, unlike `perf` PAPI is rarely installed by default.

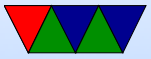
- `papi_component_avail` – list all components on system
- `papi_avail` – list all predefined events
- `papi_native_avail` – list all native events



PAPI Instrumentation

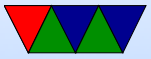
Code has to be instrumented and linked against PAPI library.

Usually this is done manually, but some tools can do this automatically via binary instrumentation.

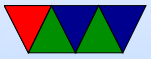


PAPI Timers

```
#include "papi.h"

int main(int argc, char **argv)  {
      int retval;
      long long start_real_usecs,end_real_usecs;
      long long start_virt_usecs,end_virt_usecs;

      retval = PAPI_library_init(PAPI_VER_CURRENT);
      if (retval != PAPI_VER_CURRENT) {
         fprintf(stderr,"Wrong␣PAPI␣version\n");
      }
      start_real_usecs = PAPI_get_real_usec();
      start_virt_usecs = PAPI_get_virt_usec();

      naive_matrix_multiply(0);
```

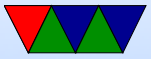


```
    end_real_usecs = PAPI_get_real_usec();
    end_virt_usecs = PAPI_get_virt_usec();

    printf("Elapsed real: %lld\n",
           end_real_usecs-start_real_usecs);
    printf("Elapsed virt: %lld\n",
           end_virt_usecs-start_virt_usecs);

    return 0;
}
```



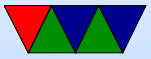
PAPI_get_real_usec() vs PAPI_get_virt_usec()

- PAPI_get_real_usec()
  wall-clock time
  maps to `clock_gettime(CLOCK_REALTIME)`
- PAPI_get_virt_usec()
  only time process is actually running
  maps to `clock_gettime(CLOCK_THREAD_CPUTIME_ID)`



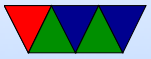
Measuring Predefined Event

- We'll use the `PAPI_TOT_INS` pre-defined counter
- On Sandybridge this maps to INSTRUCTION_RETIRED
- Currently PAPI can have more elaborate pre-defined events than perf (can do linear combinations, etc).

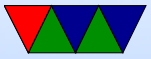


PAPI_TOT_INS Measurement

```
#include "papi.h"

int main(int argc, char **argv) {

        int retval,event_set=PAPI_NULL;
        long long count;

        retval = PAPI_library_init(PAPI_VER_CURRENT);
        if (retval != PAPI_VER_CURRENT)
            fprintf(stderr,"Wrong␣PAPI␣version\n");
        retval = PAPI_create_eventset( &event_set);
        if (retval != PAPI_OK)
            fprintf(stderr,"Error␣creating␣eventset\n");
        retval = PAPI_add_named_event( event_set,
                                       "PAPI_TOT_INS" );
```

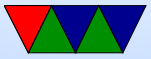


```
    if (retval != PAPI_OK)
        fprintf(stderr,"Error adding event\n");
    retval = PAPI_start(event_set);

    naive_matrix_multiply(0);

    retval = PAPI_stop(event_set,&count);
    printf("Total instructions: %lld\n",count);

    return 0;
}
```



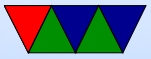
Results

```
vince@vincent-weaver-1:~/class$ ./matrix_multiply.papi
Matrix multiply sum: s=27665734022509.746094
Total instructions: 945573824
```



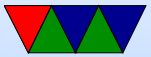
PAPI Overflow

- PAPI Can do overflow, but only provides RAW Program Counter
- Need external tool if want more detailed info

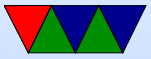


Performance Measurement Methodologies

- Aggregate Count – overall, total counts
- Profiling – measure at beginning of function.
  gprof, Valgrind Callgrind.
  records *every* entry/exit into a function, knows full backtrace
- Statistical Sampling – samples at rate of some sort of periodic counter
  perf record, oprofile

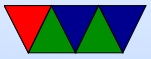


can completely miss important functions if unlucky, harder to get backtrace

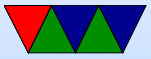


Inclusive vs Exclusive

- Exclusive only lists time spent in actual function
- Inclusive includes time spent in all child functions



Counter Overhead



Counter Accuracy