Modern CPU Performance Analysis on Linux

Andi Kleen

Intel Corporation

June 2011

My first task in the Real World was to read and understand a 200,000 line Fortran program, then speed it up by a factor of two.

- Ed Post, Real programmers don't use Pascal

Tools

- oprofile
- perf
- perf wrappers: ocperf, perf+libpfm etc.
- simple-pmu

Basics

- Sampling: oprofile, perf top, perf record/report
- Counting: perf stat, simple-pmu

Basic operation of a PMU

- Number of counters in the CPU
- or other devices like GPU, NIC, Uncore
- Event code to enable counting for an event
- Interrupt when threshold reached: allows sampling
- Modern PMUs have additional features

Classic profiling

- Follow the Pareto principle 80-20 (or 90-10 according to Knuth)
- Use sampling with cycle counter
- Identify hottest code
- Improve
- Repeat

Problems

- 90 percent of time in 10 percent of the code
- Flat profile

Skid (or “instruction shadow”)

- Sampling inaccurracy
- When sampling interrupt does not occur on the exact instruction that caused the event.
- But a few instructions later.
- Can make it hard to analyze code.
- Modern PMUs have (limited) ways around it

Standard CPU events in perf

```
% perf list
...
  cpu-cycles OR cycles                [Hardware event]
  stalled-cycles-frontend             [Hardware event]
  stalled-cycles-backend              [Hardware event]
  instructions                        [Hardware event]
  cache-references                    [Hardware event]
  cache-misses                        [Hardware event]
  branch-instructions                 [Hardware event]
  branch-misses                       [Hardware event]
  bus-cycles                          [Hardware event]
...
```

Raw events.

```
% perf list
 ...
 rNNN[:EEE] (see 'perf list --help' on how to encode it)
                                [Raw hardware event descriptor]
```

Standard perf methood: not user friendly

Example:

If the Intel docs for a QM720 Core i7 describe an event as:

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
A8H	01H	LSD.UOPS	Counts the number of micro-ops delivered by loop stream detector	Use cmask=1 and invert to count

raw encoding of 0x1A8 can be used:

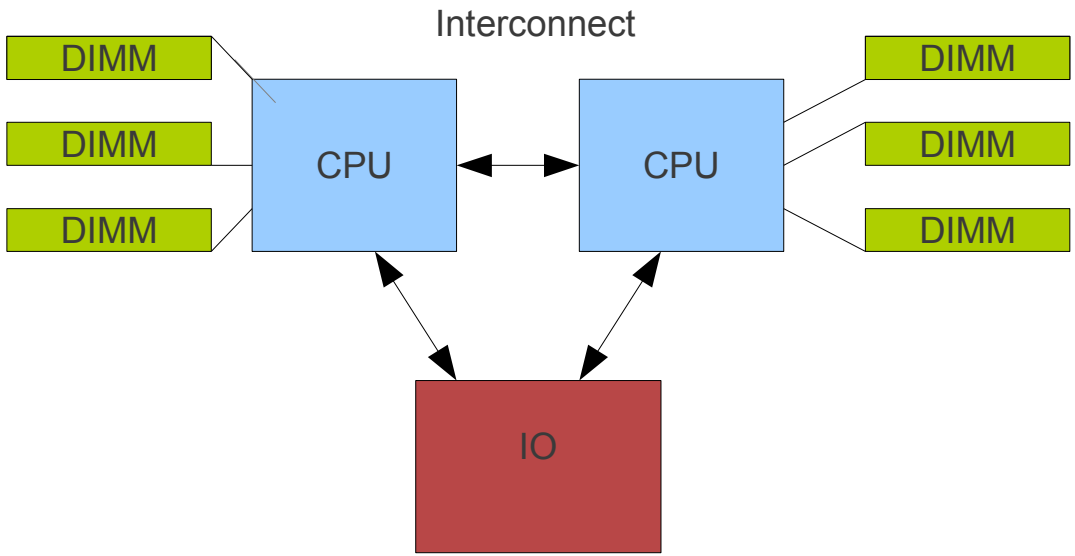
```
perf stat -e r1a8 -a sleep 1
perf record -e r1a8 ...
```

ocperf

- Perf wrapper to support Intel specific events
- Allows symbolic events and some additional events

ocperf demo

Simple NUMA system



Cache hierarchy

example, not necessarily a real system

1x	L1 cache
6x	L2 cache
60-70x	L3 cache (LLC)
180x	remote cache
200x	local DRAM
300+x	remote DRAM (NUMA)

Optimization goal:

- optimize access pattern to avoid expensive misses
- Avoid remote misses on NUMA systems.
- And cache misses everywhere
- Profile for specific misses

Generic cache events in perf

L1-dcache-loads
L1-dcache-load-misses
L1-dcache-stores
L1-dcache-store-misses
L1-dcache-prefetches
L1-dcache-prefetch-misses
L1-icache-loads
L1-icache-load-misses
L1-icache-prefetches
L1-icache-prefetch-misses
LLC-loads
LLC-load-misses
LLC-stores
LLC-store-misses
LLC-prefetches
LLC-prefetch-misses

dTLB-loads
dTLB-load-misses
dTLB-stores
dTLB-store-misses
dTLB-prefetches
dTLB-prefetch-misses
iTLB-loads
iTLB-load-misses
branch-loads
branch-load-misses

NUMA profiling

- Understand where remote memory access happens.
- offcore events profile accesses outside a CPU core
- Can be only profiled with ocperf currently!

```
(excerpt from >420)
offcore_requests.any
offcore_requests.any.read
offcore_response.any_data.any_cache_dram_0
offcore_response.any_data.any_llc_miss_0
offcore_response.any_data.remote_dram_0
offcore_response.any_data.local_cache_0
...
```

Example
Profile total and remote DRAM accesses

```
ocperf record -a -e offcore_response.any_data.remote_dram_0 sleep 10
<start a kernel compile>
ocperf report
# Events: 1K offcore_response.any_data.remote_dram_0
#
# Overhead          Command                Shared Object
Symbol
# ........  ...............  ........................  ..............
#
    10.24%               as  [kernel.kallsyms]         [k] copy_page_c
     8.09%               as  ld-2.10.1.so              [.] 0xb22a
     8.03%             make  make                      [.] 0x1b478
     7.08%              cc1  ld-2.10.1.so              [.] 0x1590d
     5.87%              cc1  cc1                       [.] 0x581cf4
     4.66%              cc1  [kernel.kallsyms]         [k] copy_page_c
     3.29%              cc1  [kernel.kallsyms]         [k] copy_user_g
     3.03%               sh  [kernel.kallsyms]         [k] copy_page_c
```

Profiling NUMA behavior of IO devices

- Not counted by uncore core event (only CPU accesses)
- Can be counted by uncore profiling driver.
- Work in progress

Load latency

- Profile loads by latency
- Provide the addresses and statistics
- Precise – avoids skid.
- But needs some way to map back addresses
- Out of tree feature currently, see references

Load latency example

```
% perf lat –L 200 record ./touchmem 10M
% perf lat report
Data source statistics
======================
                    L1–local: total latency=    2105, count=       4 (avg=526)
                    L2–snoop: total latency=    2500, count=       6 (avg=416)
...
    L3–miss, remote, exclusive: total latency=     281, count=       1 (avg=281)


Data linear address statistics
==============================
             ffff88006eaadb00: total latency=    1011, count=       1 (avg=1011)
             ffffffff81ca7228: total latency=     746, count=       2 (avg=373)
             ffffea00022230a8: total latency=     600, count=       1 (avg=600)
             ffffea0001fa48f0: total latency=     408, count=       1 (avg=408)
                 7f19dd12a8a0: total latency=     392, count=       1 (avg=392)
                 7f19dcdde9c6: total latency=     379, count=       1 (avg=379)
                 7f19dcddfbb4: total latency=     376, count=       1 (avg=376)
             ffff88006eaadae0: total latency=     348, count=       1 (avg=348)
             ffffea00009fc03c: total latency=     345, count=       1 (avg=345)
             ffff8801316ffd60: total latency=     281, count=       1 (avg=281)
```

Different problem

- Measuring code inside a program
- Need fast way to access cycles.

Measuring cycles of code - classical x86 way

```
a = rdtsc();
for (i = 0; i < 100; i++)
      my_function();
b = rdtsc();
printf("cycles %llu\n", (b - a)/100);
```

Or better without loop overhead

```
#define R5(x)    x; x; x; x; x
#define R25(x)   R5(x); R5(x); R5(x); R5(x); R5(x)
#define R100(x) R25(x); R25(x); R25(x); R25(x)

static inline void mycode()
{
    ...
}

a = rdtsc();
R100(mycode() ...);
b = rdtsc();
printf("cycles %llu\n", (b - a)/100);
```

Problems of RDTSC

- Designed to be a constant clock
- Frequency constant but unclear:
  - Hard to find current frequency
  - CPU frequency changes with cpufreq and Turbo mode
  - But RDTSC frequency stays the same (and is unclear)

Synchronization

RDTSC does not synchronize out of order CPU.

Can cause inaccuracy in cycle measurement.

```
400876:   0f 31              rdtsc
400878:   48 89 d1           mov    %rdx,%rcx
40087b:   89 c0              mov    %eax,%eax
40087d:   48 c1 e1 20        shl    $0x20,%rcx
400881:   48 09 c1           or     %rax,%rcx
400884:   4d 85 e4           test   %r12,%r12
400887:   7e 13              jle    40089c
400889:   4c 89 e8           mov    %r13,%rax
40088c:   0f 1f 40 00        nopl   0x0(%rax)
400890:   c6 00 01           movb   $0x1,(%rax)
400893:   48 83 c0 01        add    $0x1,%rax
400897:   48 39 d8           cmp    %rbx,%rax
40089a:   75 f4              jne    400890
40089c:   0f 31              rdtsc
```

Synchronized

```
#define R5(x)    x; x; x; x; x
#define R25(x)   R5(x); R5(x); R5(x); R5(x); R5(x)
#define R100(x) R25(x); R25(x); R25(x); R25(x)

sync_core();
a = rdtsc();
R100(mycode() ...);
b = rdtsc();
sync_core();
printf("cycles %llu\n", (b - a)/100);
```

Synchronized with RDTSCP
Only works on recent CPUs! (grep rdtscp /proc/cpuinfo)

```
#define R5(x)    x; x; x; x; x
#define R25(x)   R5(x); R5(x); R5(x); R5(x); R5(x)
#define R100(x) R25(x); R25(x); R25(x); R25(x)

a = rdtscp();
R100(mycode() ...);
b = rdtscp();
printf("cycles %llu\n", (b - a)/100);
```

Frequency problem still there

simple-pmu

- Simple kernel driver that starts fixed Intel counters
- Enable them for fast ring 3 read with RDPMC
- Returns current cycles in whatever the frequency is

Simple PMU example

```
#include "cycles.h"

pin_cpu(NULL);
if (perfmon_available() < 0) error
sync_core();
a = unhalted_core();
R100(mycode());
b = unhalted_core();
sync_core();
printf("cycles %llu\n", (b - a)/100);
```

Also available:

- instruction_retired();
- unhalted_ref()

simple-pmu caveats

- Does not tick in idle
- Should not switch CPUs because they are unsynchronized (use pin_cpu())
- Out of tree driver

References

simple-pmu

`http://halobates.de/simple-pmu`

ocperf

`ftp://ftp.kernel.org/pub/linux/kernel/people/ak/ocperf/ocperf4.tgz`

load-latency

`http://git.kernel.org/?p=linux/kernel/git/ak/linux-misc-2.6.git;a=shortlog;h=perf/load-latency`

Intel Software Developer's Manual: Volume 3 Chapter 30 and appendix A

`http://www.intel.com/products/processor/manuals/`

Backup

oprofile

```
    % opcontrol -l
oprofile: available events for CPU type "Intel Westmere microa

See Intel Architecture Developer's Manual Volume 3B, Appendix A
Intel Architecture Optimization Reference Manual (730795-001)

...

CPU_CLK_UNHALTED: (counter: all)
        Clock cycles when not halted (min count: 6000)
INST_RETIRED: (counter: all)
        number of instructions retired (min count: 6000)
LLC_MISSES: (counter: all)
        Last level cache demand requests from this core that m
        ...
```