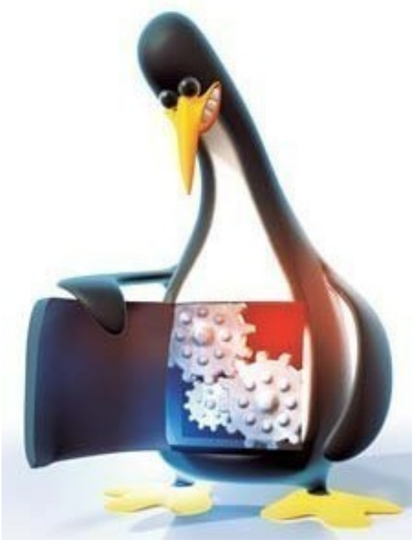
Performance Evaluation & Workload Deployment



Outline

- Performance definition
- Workloads & Benchmarks
- Static performance evaluation (Software)
- Performance counters (with Intel processors)
- Non-intrusive performance evaluation

Performance

- Two definitions
  - **Latency (execution time)**: time to finish a fixed task
  - **Throughput (bandwidth)**: number of tasks in fixed time
  - Very different: throughput can exploit parallelism, latency cannot
  - Often contradictory (latency **vs.** throughput)
    - Will already know many examples of this
  - Choose definition that matches goals (most frequently throughput)
    - Scientific program: latency; web server: throughput?

- Example: move people from A to B, 10 km
  - Car: capacity = 5, speed = 60 km/hour
  - Bus: capacity = 60, speed = 20 km/hour
  - Latency: **car = 10 min**, bus = 30 min
  - Throughput: car = 15 PPH (count return trip), **bus = 60 PPH**

Performance Improvement

- Configuration A is X times faster than Configuration B if
  - Latency(P,A) = Latency(P,B) / X
  - Throughput(P,A) = Throughput(P,B) * X
- Configuration A is X% faster than Configuration B if
  - Latency(P,A) = Latency(P,B) / (1+X/100)
  - Throughput(P,A) = Throughput(P,B) * (1+X/100)

- Car/bus example
  - Latency? Car is 3 times (and 200%) faster than bus
  - Throughput? Bus is 4 times (and 300%) faster than car

What Is ‘P’ in Latency(P,A)?

- **Program**
  - Latency(A) makes no sense, processor executes **some program**
  - But which one?
- Actual target workload?
  - \+ Accurate
  - – Not portable/repeatable, overly specific, hard to pinpoint problems
- **Some representative benchmark program(s)?**
  - \+ Portable/repeatable, pretty accurate
  - – Hard to pinpoint problems, may not be exactly what you run
- Some small kernel benchmarks (micro-benchmarks)
  - \+ Portable/repeatable, easy to run, easy to pinpoint problems
  - – Not representative of complex behaviors of real programs

SPEC Benchmarks

- SPEC (Standard Performance Evaluation Corporation)
  - http://www.spec.org/
  - Consortium of companies that collects, standardizes, and distributes benchmark programs
  - Post **SPECmark** results for different processors
    - 1 number that represents performance for entire suite
  - Benchmark suites for CPU, Java, I/O, Web, Mail, etc.
  - Updated every few years: so companies don't target benchmarks: **benchmarketing**

- SPEC CPU 2006
  - 12 "**integer**": bzip2, gcc, perl, xalancbmk (xml), h264ref (VC), etc.
  - 17 "**floating point**": povray(openGL), namd (biology), weather, etc.
  - Written in C, C++ and Fortran

Workloads

- Single application workload
  - Mostly devoted to understand/improve/compare HW
  - Runtime dominated by the workload (+99%)
  - Analyze effectiveness at some HW piece
    - SpecCPU (CPU/Mem, single core)
    - SpecOMP, PARSEC, NPB (CPU/Mem, multicore)
    - SpecViewPerf (Graphics)
    - …

- Complex workloads
  - Higher levels workloads, multi-system
  - Analyze both HW and SW (computing system level)
    - SpecJBB (AppServer), SpecWeb (WebServer), SpecVirt (Virtualized Environement), …
    - Transaction Performance Processing Council: TPC-C/DS/E/VMS

Workloads

- Standardized
- Repeatable
- Guarantee in the results
- Complex to deploy
  - In some cases only a specification is provided (TPC)

Performance evaluation

- Useful for…
  - … proposing architectural changes (not our case)
  - … improve code or software-stack (our case)

- How is performance evaluated?
  - time < my application >
  - How in an application such as SpecWeb2005?

- How to look at the details?

Code optimization

- Find the “bottleneck in your code”!
  - Done manually or automatically, a.k.a. profile driven compilation

- GNU “**gprof**”
  1. Produce an instrumentalized code

     “**gcc -pg -g3 main.c -o a.out**”
  2. Run it (~2-10x slower)

     “**./a.out**”
  3. Analyze the results

     “**gprof -l**” (line-by-line)

     “**gprof**” (full analysis)

Gprof example

```c
#include<stdio.h>

void new_func1(void);

void func1(void)
{
   printf("\n Inside func1 \n");
   int i = 0;

   for(;i<0xffffffff;i++);
   new_func1();

   return;
}

static void func2(void)
{
   printf("\n Inside func2 \n");
   int i = 0;

   for(;i<0xffffffaa;i++);
   return;
}
void new_func1(void)
{
   printf("\n Inside new_func1()\n");
   int i = 0;

   for(;i<0xffffffee;i++);

   return;
}

int main(void)
{
   printf("\n Inside main()\n");
   int i = 0;

   for(;i<0xffffff;i++);
   func1();
   func2();

   return 0;
}
```

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  s/call   s/call  name
 33.69     11.46    11.46        1   11.46    22.90  func1
 33.63     22.90    11.44        1   11.44    11.44  new_func1
 33.63     34.35    11.44        1   11.44    11.44  func2
  0.12     34.39     0.04                             main
```

```
index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.04   34.35                 main [1]
               11.46   11.44       1/1           func1 [2]
               11.44    0.00       1/1           func2 [4]
-----------------------------------------------
               11.46   11.44       1/1           main [1]
[2]     66.6   11.46   11.44       1         func1 [2]
               11.44    0.00       1/1           new_func1 [3]
-----------------------------------------------
               11.44    0.00       1/1           func1 [2]
[3]     33.3   11.44    0.00       1         new_func1 [3]
-----------------------------------------------
               11.44    0.00       1/1           main [1]
[4]     33.3   11.44    0.00       1         func2 [4]
-----------------------------------------------
```

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
 31.29     10.64    10.64                             func2 (test.c:22 @ 400663)
 31.11     21.23    10.58                             func1 (test.c:11 @ 40062b)
 28.73     31.00     9.77                             new_func1 (test.c:30 @ 400696)
  4.90     32.67     1.67                             new_func1 (test.c:28 @ 40068f)
  2.58     33.55     0.88                             func1 (test.c:9 @ 400624)
  2.35     34.35     0.80                             func2 (test.c:20 @ 40065c)
  0.09     34.38     0.03                             main (test.c:40 @ 4006c9)
  0.03     34.39     0.01                             main (test.c:38 @ 4006c2)
  0.00     34.39     0.00        1     0.00     0.00  func1 (test.c:7 @ 400608)
  0.00     34.39     0.00        1     0.00     0.00  func2 (test.c:18 @ 400640)
  0.00     34.39     0.00        1     0.00     0.00  new_func1 (test.c:26 @ 400673)
```

```
movzx     ecx, [rax+0x2]
call      0x77ef7870
cmp       rax, rdx

jz        0x77f1eac9
```

```
movzx     ecx, [rax+0x2]
....
Instrument code
...
call      0x77ef7870
....
Instrument code
...
cmp       rax, rdx
....
Instrument code
...
jz        0x77f1eac9
....
Instrument code
...
```

Usefulness

- Feed-back Gcc with profiling information (profile guided compilation)
  - Useful to improve static scheduling (if biased execution)

```
"gcc --profile-generate -O3 main.c -o a.out"
"./out"
"gcc --profile-use -O3 main.c -o a.out"
```

  - Incorporate this in a **`Makefile`** is pretty simple and might improve your application performance

- What if I want to see the "whole picture"?
  - Instrumentation degrades performance and has very low precision
  - Some hardware effects are not captured correctly (e.g. cache misses, branch miss predictions, etc…)

Performance counters

- Modern processors have programmable counters to quantify:
  - Cache misses (at any level)
  - Instruction fetched
  - Instructions executed
  - Branches taken
  - …
  - Power consumption, temperature ! (v.gr. In some Intel Xeon)

- Does not interfere with normal execution
- At very fine granularity
- Can provide much deeper insights than instrumented code
- Can be used on-line to do “fancy things”
  - E.g. homogenize thermal map of the DC (better PUE), consolidate cores, etc…

Intel Performance Counter

- Since Sandy bridge (may change from Server to Desktop Family, i.e. Core iX and Xeons families, i.e. E, X , MP,etc…)
  - 8 programmable Counters per Core
  - 3 fixed Counters per Core
  - 2 programmable Counters for LLC Communication per Core
  - 2 programmable Counters Uncore
  - 1 fixed Counter Uncore
  - 48 bit width
  - per HW Thread Counting
  - "Precise Event Based Sampling"

- Most modern processors have performance counters too!
  - http://oprofile.sourceforge.net/docs/

- Tools to access the Perf. Cntrs.
  - Directly (ouch!)
    - Are CPU registers (addressable directly in root mode from your application)
  - System performance analysis (**system level use**)
    - Intel (PMC) Performance Counter Monitor
    - **SystemTap, oprofile, perf, Dtrace**
    - **…**
  - Optimization/debugging frameworks (**app level use**)
    - **GNU perf, oprofile**
    - Intel vTune
    - …

Linux perf

- Restricted to root
  - To others echo “0” /proc/sys/kernel/perf_event_paranoid
  - sysctl for persistent
- Accountable resources (include SW and HW)
  - perf list
- System wide performance monitor
  - perf top
- Application performance monitor (events > counters)
  - perf stat [application]
- Record only specific events, call-graphs, attach record to specific running process
  - perf record –e <events> [application]
  - perf record –e <events> -p (PID)

Events

- Predefined
  - Caches
  - CPU
  - Branches
- Vendor Specifics
  - Intel Processors (see Chapter 19 in http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3b-part-2-manual.pdf)
  - 100s of different events
  - Temperature, Energy, Different domains (CPU, Mem, motherboard, ….)
- Very specific and powerful
- Other tools can use this
  - PowerTop for power measuring
  - Guess how Android analyze power consumption…

Internal of Performance counters

- #Events > #Counters
  - Reuse counters for different events, i.e. time sharing counters
  - Rotate the use of the counters per event
    - Sampling period is important
  - Controlled with “–c”

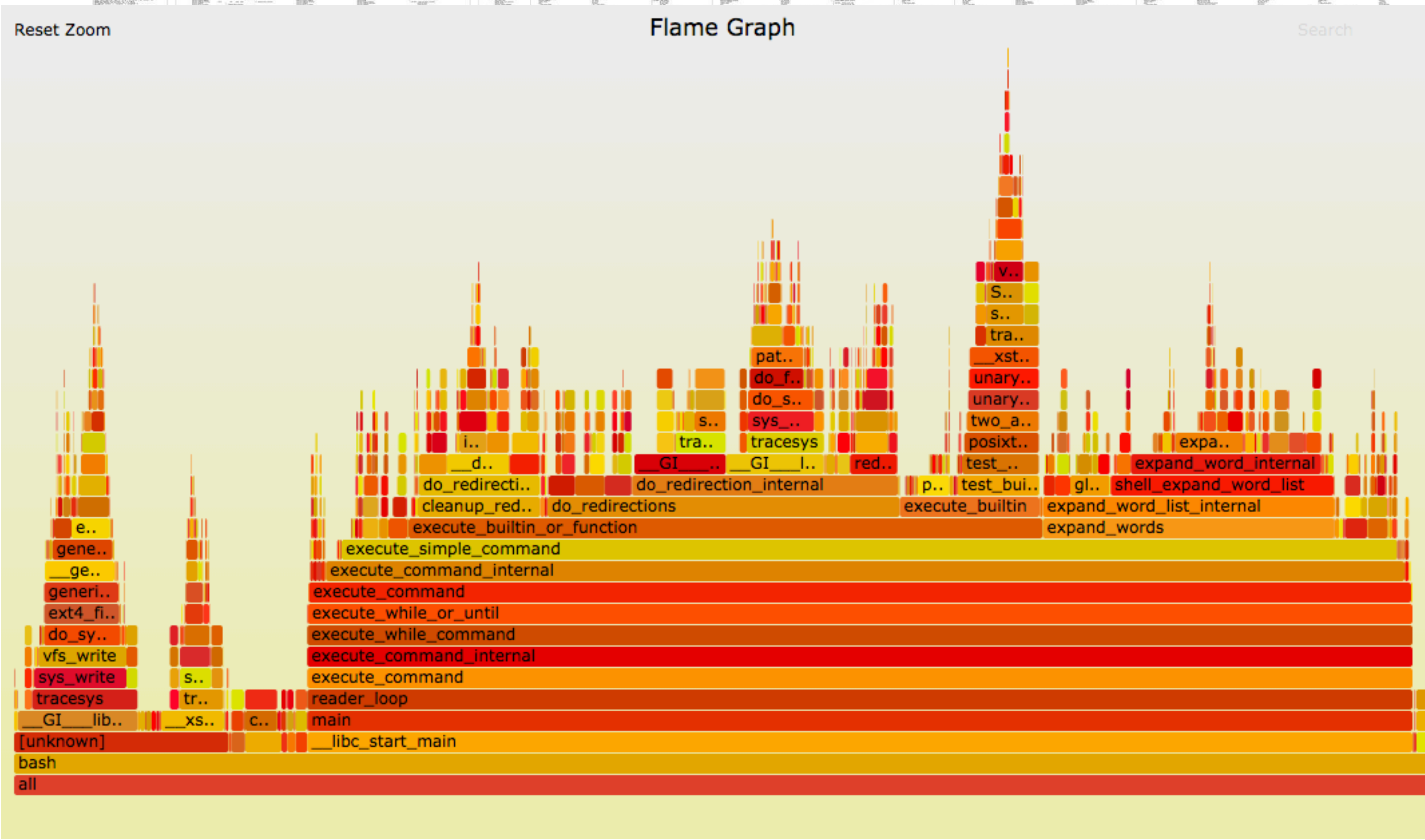
- Counters are limited (48bits)
  - Can overflow

- Performance counters might be used in a virtualized environment
  - If Hypervisor has to be enabled (usually advanced processor options: Enable CPU profiling)

Oprofile

- An alternative to the perf
- Some examples of performance optimization in
  - http://www.ibm.com/developerworks/linux/library/l-oprof/l-oprof-pdf.pdf

How to visualize the problem: Flame Graphs(http://www.brendangregg.com/)



How to monitor complex Workloads?

- Perf is too low-level for this task (even soft events)
  - e.g.: How to monitor web-server responsiveness?
  - Need to provide to a specific task and analyze what is going on
    - ~~*perf* can do this in the kernel but not in the user space~~
    - ~~*perf* is too rudimentary for complex tasks!~~

- Alternatives
  - System Tap
    - Linux
  - Dtrace
    - Solaris, Linux, OSX, BSD

SystemTap

- A higher level of abstraction over kprobes (and other performance analysis tools)
  - Scripting capabilities

- How works
  - User creates a .stp script (with all the commands required to carry out the analysis)
  - System tap creates a kernel module (.ko) file (from a previous .stp→.c conversion)
  - System tap loads the module into the kernel
  - Script reports events of interest (i.e. the user interest) thorough text or binary formats
  - Runs indefinitely or the amount of time required by the user (exits ^C)
  - Performance unintrusive

How to use SystemTap?

- Write your script

  ```
  #! /usr/bin/env stap
  probe begin { println("hello world") exit () }
  #More in http://sourceware.org/systemtap/SystemTap_Beginners_Guide.pdf
  ```

- Run it!
  - stap hello.stp

- Can do much complexes things!
  - Can trace anything in the kernel
    - CPU, Mem, Networking, Disk, ….
  - Can trace also anything in the user space
  - And it is fully programmable!
  - Much better resolution and flexibility than *vmstat*, *iostat*, *top*, *ps*, etc..

Probes: fork track

```
#!/usr/bin/stap
# Use with "stap fork.stp" to see module compilation and load
# requires kernel debug symbols
# interruptible with exit

global proc_counter #global variable

probe begin {
        print ("Fork monitoring start....Press ^C to terminate\n")
        printf ("%-25s %-10s %-s\n", "Process Name", "Process ID", "Clone Flags")
}

probe kernel.function("do_fork") { #When system call "do_fork" is called do this
        proc_counter++
        printf("%-25s %-10d 0x%-x\n", execname(), pid(), $clone_flags)
}

probe end {
        printf ("\n%d forks during the observed period\n", proc_counter)
}
```

Something more useful: intercept top 10 VFS processes

```
#! /usr/bin/env stap
#

global reads, writes, total_io

probe vfs.read.return {
   reads[execname()] += bytes_read
}

probe vfs.write.return {
   writes[execname()] += bytes_written
}

# print top 10 IO processes every 5 seconds
probe timer.s(5) {
   foreach (name in writes)
      total_io[name] += writes[name]
   foreach (name in reads)
      total_io[name] += reads[name]
   printf ("%16s\t%10s\t%10s\n", "Process", "KB Read", "KB Written")
   foreach (name in total_io- limit 10)
      printf("%16s\t%10d\t%10d\n", name,
           reads[name]/1024, writes[name]/1024)
   delete reads
   delete writes
   delete total_io
   print("\n")
}
```

See http://sourceware.org/systemtap/examples/ for more examples

A easy introductory guide in:

http://pic.dhe.ibm.com/infocenter/lnxinfo/v3r0m0/topic/liaai.systemTap/liaaisystap_pdf.pdf

Access to @perf facilities from system 2.1, i.e. you can get L2 misses for any complex workload!

Easier tracer: sysdig

- See the top processes in terms of network bandwidth usage
  - sysdig -c topprocs_net

- See all the GET HTTP requests made by the machine
  - sysdig -s 2000 -A -c echo_fds fd.port=80 and evt.buffer contains GET

- See the top processes in terms of disk bandwidth usage
  - sysdig -c topprocs_file

- See the top processes in terms of CPU usage
  - sysdig -c topprocs_cpu

- Observe ssh activity
  - sysdig -A -c echo_fds fd.name=/dev/ptmx and proc.name=sshd

  - ....

- More examples in https://github.com/draios/sysdig/wiki/Sysdig%20Examples
- Really interesting for containers