Linux Profiling and Optimization

The Black Art of Linux Performance Tuning

Federico Lucifredi
Platform Orchestra Director
Novell, INC

Novell®

0 - Rationales

System Optimization Rationales

What are we optimizing?

- Blind, pointless or premature optimization is the worst of all evils

You must know the gory details

- Yes, that means the kernel, the CPU and all the rest

How are we altering the system to measure it ?

- Heisenberg's law of profiling
- Different tools leave different prints

Tooling criteria

- availability and stability
- level of noise introduced
- multiple tools *will* be necessary, even desirable
- how does it fail ?
- ...

System's Influences on Profiling

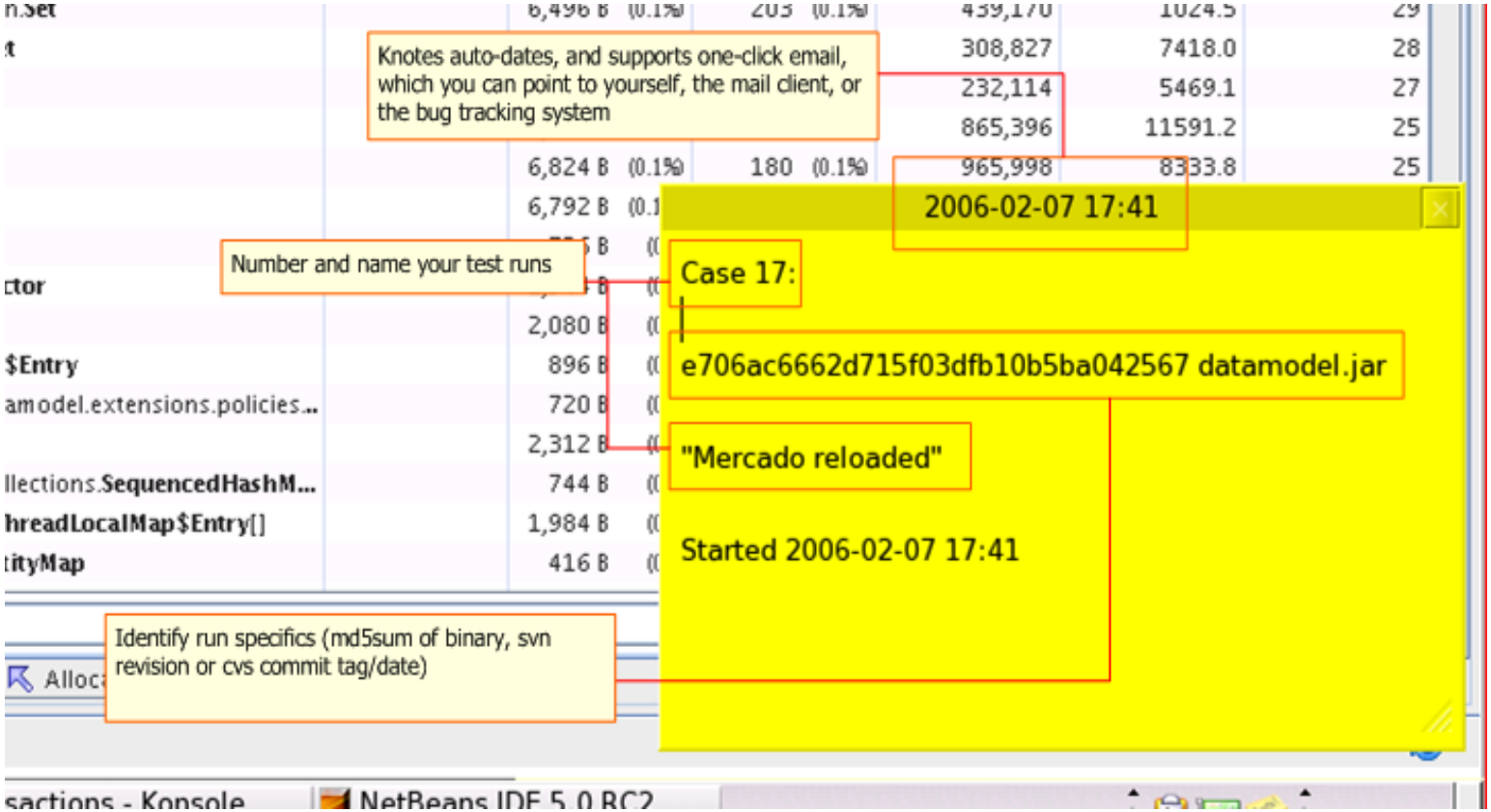
- Buffering and Caching
- Swap
- Runqueue process states
- Interrupt count
- I/O Hardware
- CPU specifics
- Memory architecture
- VM garbage collection algorithms
- ...

Performance Review Outline

- Document your investigation
  - You *have* to prove your case
- Determine the system's baseline performance
  - Start with a clean-slate, no-load system
- Determine the actual performance of X
  - How is the system performing with current version of X ?
- Track Down Obvious Bottlenecks
  - running out of ... ?
- Establish *realistic* target metric
  - Optimize *only* what necessary. Leave aesthetics to a refactoring
  - Time (YOURS) is the other constraining factor

Testing Methodology

- Document your investigation (Again!)
  - You *will* forget



Testing Methodology

- Document your investigation (Again!)
  - You *will* forget
- Change only one thing at a time
  - You are smart, but not *that* smart
- Verify your conjecture
  - you need tangible proof – cold hard facts
- Check again
  - with another tool – the one you just used lied to ya
- Be **very** patient
  - This is hard, frustrating work. Period.
- Use a process
  - Others need to know you have not escaped the country (yet)

Profiling Objectives

- Don't *ever* optimize everything
  - You can't – conflicting aims.
- Be clear on purpose
  - Optimization
    - memory, permanent storage, net thruput, speed, etc
  - Debugging
    - memory leaks
  - Testing
    - coverage, load testing
- Choose a clear objective
  - How many transactions ? How little RAM? How many clients ?
- Determine acceptable tradeoffs
  - This is obvious to you, make sure it is to those above you

Tools Lie

- Interfering with normal operation
  - You are running on the system you are observing
  - You are instrumenting the OS or the Binary
  - Even hardware emulators have limits - and anything else is *worse*
- Common results
  - Incomplete information by nature is misleading
  - Complete information is very obtrusive
  - Incorrect or corrupted information is often produced (sic)
  - Footprint size varies
  - Failure modes often dependent on technology used
  - Out-of-scope data misleading and/or overwhelming
- Verify your results
  - the odd as well as the actionable

I – Gory Details

Hardware Architecture: RAM

- Speedy storage medium
  - Orders of magnitude faster (or slower?)
    - *every* application running on a 7th gen CPU has a bottleneck in RAM fetching
    - whenever pre-fetching fails, things get even worse (instruction optimization)
    - you can hand-tune prefetching (*_mm_prefetch),* 100 clocks in advance
  - Natural-order access problems
    - paging, swapping and generally all large-RAM access scenarios
- Less is more
  - you have oodles of RAM, but using less of it helps caching
    - choosing cache-friendly algorithms
- Size buffers to minimize eviction
- Optimize RAM use for cache architecture
  - not everything needs to be done by hand ->Intel compiler

Hardware Architecture: The Cache

- Helping RAM to catch up
  - Different architectures in different generations
    - P4s have L1(data only), L2(data+instructions)(30 times larger, 3 times slower)
    - optional L3
    - Different caching algorithms make exactly divining the caching behavior **hard**
      - but there is still hope
    - whenever pre-fetching fails, things get even worse (instruction optimization)
    - you can hand-tune prefetching (*_mm_prefetch),* 100 clocks in advance
  - Locality could be a problem in this context
    - p4 caches by address!
    - L1 cache 64B lines, 128 lines = 8192 Bytes
    - Alignment also relevant

Hardware Architecture: The CPU

- Optimization strategies for x86 families:
  - Completely different in different generations
    - i386 (1985): selecting assembly instructions to hand-craft fast code
    - i486 (1989): different instruction choice, but same approach
    - Pentium (1993): U-V pairing, level 2 cache, MMX instructions
    - sixth generation: minimizing data dependencies to leverage n-way pipelines
      - cache more important than ever
      - SIMD
      - 4:1:1 ordering to leverage multiple execution units
    - Pentium 4:
      - hardware prefetch
      - branching errors
      - data dependencies (micro-ops, leverage pipelines)
      - thruput and latency
    - and you still have hyperthreading to be added in this picture....
  - The Intel compiler lets you define per-arch functions

Hardware Architecture: More CPU

- Instructions trickery
  - loop unrolling
    - contrasting objectives
    - keep it small (p4 caches decoded instructions)
  - Important: branch prediction
    - increasing accuracy of branch prediction impacts cache
    - reorder to leverage way processor guesses (most likely case first)
    - remove branching with CMOV, allow processor to go ahead on both paths
    - Optimize for long term predictability, not first execution
    - a missed branch prediction has a large cost (missed opportunity, cache misses, resets)
  - SIMD and MMX
    - Sometimes we can even get to use them
  - The compiler can help a bit
    - check options
  - Pause (new)
    - delayed NOOP that reduces polling to RAM bus speed (*__mm_pause*)

Hardware Arch: Slow Instructions

- Typical approaches:
  - Precomputed table
    - maximize cache hits from table
    - keep it small (or access it cleverly)
  - Latency, thruput concerns
    - find other work to do
  - Fetching
    - top point to optimize
  - Instructions that cannot be executed concurrently
    - Same execution port required
  - The floating point specials:
      - Numeric exceptions
      - denormals
      - integer rounding (x86) vs truncation (c)
      - partial flag stalls
    - and you still have hyperthreading to be added in this picture....

Kernel Details

- Caching and buffering
  - This is good – but you need to account for it in profiling (also: swap)
  - All free memory is used (released as needed)
    - use slabtop to review details
    - behavior can be tuned (/proc/sys/vm/swappiness)
- I/O is the most painful point
  - adventurous ? you can tune this via kernel boot params
  - factors:
    - latency vs thruput
    - fairness (to multiple processes)
  - you can also tune the writing of dirty pages to disk
- Linux supports many different filesystems
  - different details, valid concern in *very* specific cases
  - typically layout much more important

II – State of the machine

What is in the machine: hwinfo

hwinfo

- too much information – add double-dash options
- great for support purposes, but also to get your bearings

- syntax: hwinfo [options]

- example: hwinfo --cpu

CPU load: top

top

- top gives an overview of processes running on the system
- A number of statistics provided covering CPU usage:
  - us : user mode
  - sy : system (kernel/)
  - ni : niced
  - id : idle
  - wa : I/O wait (blocking)
  - hi : irq handlers
  - si : softirq handlers
- Additionally, the following are displayed:
  - load average : 1,5,15-min load averages
  - system uptime info
  - total process counts (total|running|sleeping|stopped|zombie)
- syntax: top [options]

CPU load: vmstat

vmstat

- Default output is *averages*
- Stats relevant to CPU:
  - in : interrupts
  - cs : context switches
  - us : total CPU in user (including “nice”)
  - sy : total CPU in system (including irq and softirq)
  - wa : total CPU waiting
  - id : total CPU idle
- syntax: vmstat [options] [delay [samples]]

- *remember: not all tools go by the same definitions*

CPU: /proc

/proc

- /proc knows everything
  - /proc/interrupts
  - /proc/ioports
  - /proc/iomem
  - /proc/cpuinfo

procinfo

- parses /proc for us
- syntax: procinfo [options]

mpstat

- useful to sort the data on multicore and multiprocessor systems
- syntax: mpstat [options]

CPU: others

sar

- System activity reporter
  - still low overhead

oprofile

- uses hardware counters in modern CPUs
- high level of detail
  - cache misses
  - branch mispredictions
- a pain to set up (kernel module, daemon and processing tools)

RAM: free

free

- reports state of RAM, swap, buffer, caches
- 'shared' value is obsolete, ignore it

/proc

- /proc knows everything
  - /proc/meminfo
  - /proc/slabinfo

slabtop

- kernel slab cache monitor
- syntax: slabtop [options]

RAM: others

vmstat, top, procinfo, sar

- some information on RAM here as well

I/O: vmstat

vmstat

- i/o subsystem statistics
  - D : all
  - d : individual disk
  - p : partition
- output
  - bo : blocks written (in prev interval)
  - bi : blocks read (in prev interval)
  - wa : CPU time spent waiting for I/O
  - (IO: cur) : total number of I/O ops currently in progress
  - (IO: s) : number of seconds spent waiting for I/O to complete
- syntax: vmstat [options] [delay [samples]]

I/O: iostat

iostat

- i/o subsystem statistics, but better than vmstat
  - d : disk only, no CPU
  - k : KB rather than blocks as units
  - x : extended I/O stats
- output
  - tps : transfers per second
  - Blk_read/s : disk blocks read per second
  - Blk_wrtn/s : disk blocks written per second
  - Blk_read : total blocks read during the delay interval
  - Blk_wrtn : total blocks written during the delay interval
  - rrqm/s : the number of reads merged before dispatch to disk
  - wrqm/s : the number of writes merged before dispatch to disk
  - r/s : reads issued to the disk per second
  - w/s : writes issues to the disk per second
  - rsec/s : disk sectors read per second

I/O: iostat (continued)

iostat

- output (continued)
  - wsec/s : disk sectors written per second
  - rkB/s : KB read from disk per second
  - wkB/s : KB written to disk per second
  - avgrq-sz : average sector size of requests
  - avgqu-sz : average size of disk request queue
  - await : the average time for a request to be completed (ms)
  - svctm : average service time (await includes service and queue time)

- syntax: iostat [options] [delay [samples]]

- tip: to look at swapping, look at iostat output for relevant disk

I/O: others

sar

- can also pull I/O duty

hdparm

- great way to corrupt existing partitions
- also, great way to make sure your disks are being used to their full potential

bonnie

- non-destructive disk benching
- caching and buffering impacts results in a **major** way
  - can be worked around by using sufficiently large data sizes

Network I/O

iptraf

- there are several other tools, but iptraf is considerably ahead
- somewhat GUI driven (in curses)

- syntax: iptraf [options]

III – Tracking a specific program

The Most Basic Tool: time

time

- time measures the runtime of a program – startup to exit
- Three figures provided
  - real
  - user
  - sys
- Only *user* is system-state independent, really
- syntax: time <program>

CPU load: top - II

Per-process stats

- PID : pid
- PR : priority
- NI : niceness
- S : Process status (S | R | Z | D | T)
- WCHAN : which I/O op blocked on (if any) , aka sleeping function
- TIME : total (system + user) spent since spent since startup
- COMMAND : command that started the process
- #C : last CPU seen executing this process
- FLAGS : task flags (sched.h)
- USER : username of running process
- VIRT : virtual image size
- RES : resident size
- SHR : shared mem size
- ...

CPU load: top - III

Per-process stats *(continued)*

- %CPU : CPU usage
- %MEM : Memory usage
- TIME+ : CPU Time, to the hundredths
- PPID : PID of Parent Process
- RUSER : real username of running process
- UID : user ID
- GROUP : group name
- TTY : controlling TTY
- SWAP : swapped size
- CODE : code size
- DATA : (data + stack) size
- nFLT : page fault count
- nDRT : dirty page count
-

System calls: strace

strace

- a few selected options:
  - c : profile
  - f, F : follow forks, vforks
  - e : qualify with expression
  - p : trace PID
  - S : sort by (time | calls | name | nothing). default=time
  - E : add or remove from ENV
  - v : verbose
- syntax: strace [options]
- No instrumentation used on binary, kernel trickery on syscalls

Library calls: ltrace

ltrace

- a few selected options:
  - c : profile
  - o : save output to <file>
  - p : trace PID
  - S : follow syscalls too, like strace
- syntax: strace [options]

- the -c option is a favorite quick&dirty profiling trick

Library calls: runtime loader

ld.so

- dynamic linker can register information about its execution
- environment controlled
  - libs display library search paths
  - reloc display relocation processing
  - files display progress for input file
  - symbols display symbol table processing
  - bindings display information about symbol binding
  - versions display version dependencies
  - all all previous options combined
  - statistics display relocation statistics
  - unused determined unused DSOs

- example: env LD_DEBUG=statistics LD_DEBUG_OUTPUT=outfile kcalc

profiling: gprof and gcov

gprof

- the gnu profiler
  - requires instrumented binaries (compile time)
  - gprof used to parse output file

gcov

- coverage tool – uses the same instrumentation
- similar process

process RAM

/proc

- /proc/<PID>/status
  - VmSize : amount of virtual mem the process is (currently) using
  - VmLck : amount of locked memory
  - VmRSS : amount of physical memory currently in use
  - VmData : data size (virtual), excluding stack
  - VmStk : size of the process's stack
  - VmExe : executable memory (virtual), libs excluded
  - VmLib : size of libraries in use
- /proc/<PID>/maps
  - use of virtual address space

memprof

- graphical tool to same data

Valgrind

valgrind

- emulates processor to application, providing:
  - really slow execution!
  - memcheck
  - uninitialized memory
  - leaks checking
  - VmStk : size of the process's stack
  - VmExe : executable memory (virtual), libs excluded
  - VmLib : size of libraries in use
- addrcheck
  - faster, but less errors caught
- massif
  - heap profiler
- helgrind
  - race conditions
- cachegrind / kcachegrind
  - cache profiler

Interprocess communication

ipcs

- information on IPC primitives memory in use
  - t : time of creation
  - u : being used or swapped
  - l : system wide limits for use
  - p : PIDs of creating/last user process
  - a : all information

Tips & Tricks

A few odd ones out:

- /bin/swapoff – to test (RAM permitting) w/o swap
- ps is still king when it comes to processes – look at the custom output formats for convenient scripting

IV – VM test Dummies

VM Details

- Garbage collection
  - you need to know the algorithms at play
    - floating garbage
    - stop-the world
    - compacting
    - ...
  - considerable differencies between Sun JVM and others (not to mention mono!)
    - different Vms have different profiling APIs
    - different Gcing algorithms **or implementations**
    - yes, you need to skim the relevant papers
- Profiling APIs:
  - JVMTI (current)
  - JVMPI (since 1.1) – legacy, flaky on hotspot, deprecated in 1.5
  - JVMDI (since 1.1) – legacy, also targeted at removal for 1.6

VM Tools (my favorites)

- Jprofiler (commercial)
  - extremely detailed info
    - disables minor garbage collections
    - gathers so much data it can hang itself
    - best tool out there in terms of breadth, and first choice to analyze a leak
    - deadlock-finding tools
    - easier to break, as it is doing so much more
- Netbeans profiler
  - positively great, hats off to Sun, started using it in beta
    - much less information than jprofiler, but can redefine instrumentation **at runtime (!)**
    - telemetry has zero footprint, and can be used to identify moment for a snapshot
    - generation information provides great hints for really small leaks (bad signal/noise)
    - extremely stable
- Roll-your-own

V - Conclusion

What more to say ?

There are a lot of tools out there, start picking them up and go get your hands dirty!

Resources:

Books

- Code Optimization: Effective Memory Usage (Kris Karspersky)
- The Software Optimization Cookbook (Richard Gerber)
- Optimizing Linux Performance (Philip Ezolt)
- Linux Debugging and Performance Tuning (Steve Best)
- Linux Performance Tuning and Capacity Planning (Jason Fink and Matthew Sherer)
- Self Service Linux (Mark Wilding and Dan Behman)
- Graphics Programming Black Book (Michael Abrash)

Other

- Kernel Optimization / Tuning (Gerald Pfeifer - *Brainshare*)

Any Questions ?

<flucifredi@acm.org>

Thanks for coming!

PICASso

Program in Integrative Information, Computer and Application Sciences

Princeton University

Princeton, February 2007

Novell®