

Tuning and Profiling (particularly on Unix Systems)

- Goals
- Preliminaries
- Analysis Strategy
- System Bottleneck
- Application Profiling
- Tuning Tactics

1

Goals

Possible Goals

- Decrease execution time for a task
- Increase throughput (total amount of work/time period) for a set of tasks
- Decrease latency
- Decrease systems resource use (aka. 'hogging', 'bloat')
- Graceful degradation under load

Note: *These are not equivalent. E.g., parallelizing decreases execution time & **decreases** throughput.*

-Decreasing execution time is the most common goal, but is not always the most desirable. For example, decreasing execution time may require more system resources, leading to decreased total throughput.

-Latency is critical in *real-time* systems such as hardware controllers. Latency is also important in UI implementation, where latencies above critical thresholds (approximates .1, 1 and 10 seconds) can interfere with the end users ability to focus upon, and smoothly operate the UI.

-Bloat may not affect either execution time or actual throughput if system resources are plentiful. However, bloat may reduce the potential maximum throughput possible on a particular system. Bloat can also cause degradation of performance under load, when it occurs, to be more severe.

-Graceful degradation is particularly important for 'server' architecture. When a server degrades gracefully:

-Performance drops linearly (or more slowly) as load increases.

-There are no dramatic decreases in performance.

-Requests already in progress are not terminated, or unduly delayed, by later increases in load.

-New requests on a loaded system are either queued or refused in such a way that the client can automatically resubmit them later (or submit them to another server)

2

Preliminaries

- Tunable architecture
 - Do encapsulate algorithms
 - Do consider the impact of class structure on algorithms and communications
 - Do not try to optimize at architectural level
- Use Efficient Algorithms
 - Algorithmic gains beat tuning gains: Don't try to tune a bubble-sort.
 - Consider algorithmic options: time vs. space tradeoffs, optimistic algorithms, approximate solutions, randomized algorithms
- Fix Bugs First
 - Incorrect code may have radically different performance characteristics

1. On class structure: when components of a data structure that are needed in an algorithm are spread over multiple classes, it may be difficult to optimize. Use of 'friend' may help, but sometime classes need to be restructured in order for efficient algorithms to be implemented.

2. On efficient algorithms:

- Time vs. space: different data structures and algorithms may require different amounts of resources. For example, sparse matrix data structures take less memory than arrays, but require more time to compute when matrices are dense.

- Optimistic algorithms are sometimes used in databases and distributed systems. These algorithms assume that a particular operation can proceed safely, then verify, and back out of the operation if necessary.

- Randomized algorithms are designed to achieve the correct answer with high (and *known*) probability, but may sometimes fail. Approximate algorithms yield an answer that is within a known percentage of the optimal solution. Neither should be confused with *heuristics*, which are approaches that have empirical success, but which do not guarantee anything about the probability or quality of the solution produced.

3

Analysis Strategies

- Identify use-case based benchmarks
- Look for system bottlenecks
- Examine processes
- Profile applications
- Focus on small, expensive regions of code

4

Identify benchmarks

- What patterns of usage are likely to occur?
 - What are the 80% cases?
 - Are there 10% cases that have unusual patterns of data access, or unusual input?
 - Can you construct a plausible worst-case?
- Create benchmarks based on real cases
 - Use real problems for full benchmarking
 - Miniaturize real problems for quick tests

5

Look for system bottlenecks

- Usually, a single resource will be the bottleneck at a particular time:
 - CPU ☺
 - Memory
 - I/O: Graphics, Network, Disk
- Use `vmstat`, `iostat`, `netstat`, `pmstat`/`pmval` to identify bottlenecks
- Tune against the bottleneck

Performance Co-Pilot (oss.sgi.com):

-Extremely useful for system monitoring. It allows you to get at:

- Access any performance metric available in the system.
- Access snmp-based, and database performance stats as well
- Record and playback logs of information
- Unified format for collection and display of stats
- Distributed system for collecting metrics on various hosts, and summarizing across hosts
- Has facilities to set up monitoring and alarm system for out-of-resource or other performance condition
- 'man PCPIintro' to get started, use 'pmstat' for standard stats, 'pmval' for other metrics, and 'pminfo -dtF' to list available metrics

On resources:

-CPU bottlenecks are usually the most desirable, since CPU time is usually the most costly (in terms of real \$\$). If you have to have a on a \$5K system bottleneck, it shouldn't because of \$100 worth of RAM.

On tools:

- `vmstat` shows system memory activity and cpu activity, and some filesystem activity (block I/o only)
- `iostat` shows detailed disk activity, available on RH 7.1, but not RH 6.2

6

Symptoms of System Bottlenecks

- CPU Bottlenecks:
 - CPU User+Sys activity near 100%
 - no CPU idle, no CPU wait on I/O (not available on Linux)
 - If CPU %sys is high suspect inefficient use of system calls, or borderline I/o or memory bottlenecks
 - Possibly large numbers of processes in run-queue, but not necessary
- Memory Bottlenecks:
 - Severe: processes in swap queue (or wait on swap), lots of space in use (see `swap -m`), swapping activity, free memory low
 - Moderate: high context switches, high validity faults, swapping
- I/O Bottlenecks:
 - Possibly high % sys activity in CPU, high # of system calls, # interrupts
 - I/O rate high
 - Context switches, wait on I/O, or lots of processes sleeping on I/O

7

Process Level Analysis

- Purpose:
 - Find which processes are using resources
 - identify resource use of a particular process
- Tools:
 - Process/request lists: `top`, `ps`, `pmval`, `Customlog`
 - Individual process analysis: `/usr/bin/time`, `strace`
- Strategy
 - Use `top` and `ps` to identify resource hogs
 - Use `top` (or `pmlogger`) to see how system behaves over time
 - Use `Customlog` (%T) to see which HTTP requests take a long time
 - Use `time` to look at overall usage
 - Memory and cpu patterns
 - Discrepancies between CPU and wall-clock time

Customlog:

-Standard apache logs do not include duration of request. The Customlog directive can be used to create logs with this information.

-The %T has timings, I like to include Refere, User-Agent and cookies in the logs as well, in addition to the 'standard' stuff:

```
Customlog logs/detaillog "%v %h %l %u %t \"%r\" %>s %b \"%{Referer}i\" \"%{User-agent}i\" \"%{SCRIPT_FILENAME}e\" \"%{cookie}i\" %T"
```

- In apache 1.3/2.0, using "%c"/"%X" (respectively) should get you connection status (e.g. if a browser aborted in the middle), but this doesn't work in our version.

Caveats:

- Customlog connection status flag seems broken in our version of Apache. So cannot tell if request completed early because the browser aborted. Treat anything above 300 secs with suspicion.

-default time may be shell's builtin – always specify the path

-Output for 'time' that describes # of page faults, really measures total activity on the system during the processes execution. This activity *may not have been caused by the process*. Use `pmval` instead.

Tips:

- `top` allows you to interactively sort by total memory use, cpu use, etc.

8

Profiling Overview

- Profiling = analyzing pattern of resource use within a particular program.
- Most tools focus on subroutine-level CPU use
- Other levels (instruction, system call, line, call-stack) possible
- Other resources (memory, disk, network) sometimes available
- Three gotchas:
 - Unrealistic benchmarks
 - Omitted resources
 - Heisenberg's principle

9

Types of Profiling

- Wall clock: comprehensive, bottom line
Cons: not isolated from system activity
- Analytical: precise, isolated...
Cons: changes code, typically doesn't include any system latencies, may not include system calls
- Statistical/Interrupt Driven: no code changes, captures cache/RAM latency
Cons: does not include latencies that cause process to be in 'sleep/blocked/swapped' state
- Tools: strace (system calls level), drprof/benchmark (perl), hprof (java), gprof (C,C++), Rprof (R)

•On Strace: Note that strace can be used to (roughly) profile pretty much anything, but does not give the very useful subroutine analysis.

•Additional profiling tools abound:

•Commercial: *Rational* software has some (reputedly) good ones, including *Purify* for memory profiling

•Some open source ones that may be useful:

•squidalyzer: analyze squid logs. Available from freshmeat.

•JMP: java method profiler. Available from freshmeat.

•RUE: Resource utilization explorer. Specifically for profiling servlets. Available from sourceforge.

10

Identify Small & Expensive Regions of Code

- To double the speed of a program, the code you tune has to be > 50% of the execution time
- 'Optimizing' large amounts of code is ineffective and hard to maintain
- If code takes only a small %age of run time, there is risk of 'overtuning' (tuning to the specific situation in a way that does not have general benefits)

11

Tuning Strategies

- General:
 - Waste not want not...
 - Put off until tomorrow...
 - Pre-processing
 - Efficient/Approximate/Randomized/Optimistic algorithms
 - Customized Data structures
- CPU
 - Compiler optimizations
 - Cache size
 - Arithmetic precision
 - Inlining code
- I/O
 - Correct order
 - Correct chunk size
 - Locality of reference
 - Prefetching

12

Example (step 1: benchmark)

- Report that analysis of drug abuse study was 'slow'
- Formulates some possible cases:
 - Select a {1,10,100,1000} variables, and {no, simple criteria, complex} row selection criteria on {10MB,100MB,1GB} dataset for {descriptives, boxplots, q-q plots, tab-delimited, stata, R}
- Clocked a simple case {1 var, 100MB, no rows, descriptives}
 - With a stopwatch: 5 minutes and browser timed out
 - Apache logs for request showed longer execution time (then abort)

```
vdc-prod-hmdc-harvard.edu:341.154.67.103 -- [17Nov2001:02:24:04 -0500] POST
/VDC/DSB/GA1/Disseminate HTTP/1.1 "200 766 Times/View=
prod-hmdc-harvard.edu/VDC/Study/Analyze/Submit.jsp?time=vdc-beans.RepositoryAccessDO&put=http://
stat.shedden.org/PODCE/FPUB/L/1/02755&id=field1" Mozilla/4.0 (compatible; MSIE 5.5; Windows 98)
"/home/httpd/cgi-bin/univar03.cgi" 2%SESSIONID=fq3vktqpl;
vdcCookie=vdcSession=fq3vktqpl,session,au=vdc_prod,ov=vdc 1550
```

Note: This example is cooked – the problem had already been fixed on vdc-prod, so I reproduced on login. Some of the exact numbers may vary, but the pattern is still the same

13

Example (step 2.1: bottleneck)

'vmstat' shows that CPU is bottleneck, although interrupts and context switches are suspiciously high as well:

```
[maltman@login ~/rttest]$ vmstat -n 5
procs  r b w  swpd  free  buff  cache  si  so  bi  bo  in  cs  us  sy  id
1 0 0  60  2064 111036 489164  0  0  45  1  38  19  3  1  41
1 0 0  60  2052 111036 489164  0  0  0  1  121  101 100  0  0
1 0 0  60  2052 111036 489164  0  0  0  1  121  101 100  0  0
1 0 0  60  2052 111036 489164  0  0  0  1  120  98 100  0  0
...
[maltman@login ~/rttest]$ netstat -iac
Kernel Interface table
Iface  MTU  Met  RX-OK  RX-ERR  RX-DRP  RX-OVR  TX-OK  TX-ERR  TX-DRP  TX-OVR  Flg
eth0  1500  0  12284461  0  0  0  57704048  0  0  0  BRU
lo    3924  0  896898  0  0  0  896898  0  0  0  IRU

Iface  MTU  Met  RX-OK  RX-ERR  RX-DRP  RX-OVR  TX-OK  TX-ERR  TX-DRP  TX-OVR  Flg
eth0  1500  0  12284464  0  0  0  57704049  0  0  0  BRU
lo    3924  0  896898  0  0  0  896898  0  0  0  IRU
```

14

Example (step 2.2: bottleneck)

'netstat' shows no significant activity or errors:

```
[maltman@login ~/rttest]$ netstat -iac
Kernel Interface table
Iface  MTU  Met  RX-OK  RX-ERR  RX-DRP  RX-OVR  TX-OK  TX-ERR  TX-DRP  TX-OVR  Flg
eth0  1500  0  12284461  0  0  0  57704048  0  0  0  BRU
lo    3924  0  896898  0  0  0  896898  0  0  0  IRU

Iface  MTU  Met  RX-OK  RX-ERR  RX-DRP  RX-OVR  TX-OK  TX-ERR  TX-DRP  TX-OVR  Flg
eth0  1500  0  12284464  0  0  0  57704049  0  0  0  BRU
lo    3924  0  896898  0  0  0  896898  0  0  0  IRU
```

pvmlval shows no significant disk activity:

```
[maltman@login ~/rttest]$ pvmlval disk.all.aveq [maltman@login ~/rttest]$ pvmlval disk.all.total_bytes
metric: disk.all.aveq metric: disk.all.total_bytes
host: localhost host: localhost
semantic: cumulative counter (converting to rate) semantic: cumulative counter (converting to rate)
units: millise (converting to time utilization) units: Mbyte (converting to Kbyte / sec)
samples: all samples: all
interval: 1.00 sec interval: 1.00 sec
...

```

Netstat notes:

- netstat doesn't report delta's so you have to mentally subtract each interval to look at rate information. Use pvmlval, ifstat (from freshmeat), or watch -c netstat -ia for potentially easier to read information
- Rate of transmits/receive data is important. For transmissions on localhost this will show under the 'lo' (or 'loopback') interface line.
- You should see no errors (RX-ERR), (TX-ERR) if you do there is probably a hardware problem. Any hardware problem will probably severely effect network performance, so address it first.

Pvmlval:

- disk.all.aveq metric shows if multiple processes are competing for a particular disk
- disk.all.total_bytes metric shows the amount of bytes written
- this includes raw/character level access to disks, which is omitted by vmstat
- Other pvmlval metrics can be used to examine individual disks, partitions, other disk metrics, and other system and process metrics (see 'man pvmlval')

15

Example (step 3: processes)

I used 'top' to look at progress of request:

```
2:17pm up 121 days, 1:23, 4 users, load average: 0.62, 0.27, 0.09
313 processes: 311 sleeping, 2 running, 0 zombie, 0 stopped
CPU states: 22.7% user, 40.3% system, 0.0% nice, 36.8% idle
Mem: 1048092K av, 887652K used, 16040K free, 0K shrd, 174308K buff
Swap: 530104K av, 4848K used, 525256K free
PID USER PRI NI SIZE RSS SHARE STAT LkB kCPU MEMN TIME COMMAND
18331 postmaster 0 0 2564 2500 1864 R 0 58.4 0.2 0.03 postmaster
```

First a few seconds in postmaster (retrieving object from postgres). Then 30 minutes in R.bin:

```
2:17pm up 121 days, 1:23, 4 users, load average: 1.14, 0.42, 0.15
313 processes: 311 sleeping, 0 running, 0 zombie, 0 stopped
CPU states: 0.0% user, 22.4% system, 39.0% nice, 38.0% idle
Mem: 1048092K av, 887652K used, 16040K free, 0K shrd, 9848K buff
Swap: 530104K av, 4848K used, 525256K free
42045K cached
PID USER PRI NI SIZE RSS SHARE STAT LkB kCPU MEMN TIME COMMAND
18336 nobody 19 19 786700 700M 1584 RM 0 96.6 53.7 30:08 R.bin
```

16

Example (step 4: single process timing)

I used `'/bin/time -v R ...'` to look at a smaller example:

```
Command being timed: "R --save"
User time (seconds): 357.04
System time (seconds): 3.53
Percent of CPU this job got: 94%
Elapsed (wall clock) time (h:mm:ss or m:ss): 6:23.02
Average shared text size (kbytes): 0
Average unshared data size (kbytes): 0
Average stack size (kbytes): 0
Average total size (kbytes): 0
Maximum resident set size (kbytes): 0
Average resident set size (kbytes): 0
Maximum resident set size (kbytes): 0
Major (compacting) page faults: 907
Minor (reclaiming a frame) page faults: 229112
Voluntary context switches: 0
Involuntary context switches: 0
Swaps: 0
File system inputs: 0
File system outputs: 0
Socket messages sent: 0
Socket messages received: 0
Signals delivered: 0
Page size (bytes): 4096
Data transfer: 0
```

- Time `-v` output shows process is CPU bound.
-No discrepancy between wall-clock and CPU time.

Note: discrepancy between wall-clock and CPU time means either:

- competing processes
- lots of I/O or memory paging

Note: bogus report of process sizes. Not sure why. Use `top/ps/ pmap` instead for this info

17

Example (step 4.1: mini-profile)

I cooked a short R test program, using a subset of the original data, timing calls, and simpler stat analysis:

```
data()
y<-read.table("dat",header=TRUE,row.names="age")
data()
summary(y$CHILDREN)
data()
q()
```

The internal timing calls alone show that 99%+ of the time was spent in `read.table()`

```
1 /bin/time -v R --save 0 8 example
2 read.table("dat",header=TRUE,row.names="age")
3 summary(y$CHILDREN)
4 data()
5 q()
summary(y$CHILDREN)
Min. 1st Qu. Median Mean 3rd Qu. Max.
 1. 5807  9999  8119  9999  9999
> data()
[1] "Wed Dec 26 14:53:44 2001"
? q()
```

18

Example (step 4.2: test)

- Pre-slicing the data using `'cut'` to produce only the variable being analyzed reduced `read.table()` time to a few seconds.

-But the real code differed from my benchmark

- the real problem was reduced from >30minutes to 5 minutes, almost all of it still in R

- summary() in R only took .1 seconds on the same data, so something else was going on in the R code

Example (step 4.3: profile)

'Rprof' showed that the `mode()` operation was taking an inordinate amount of time. We decided to eliminate that statistic.

```
% total % self
total seconds self seconds name
99.29 272.92 0.01 0.02 "univarStat"
98.74 271.40 0.04 0.12 "statMode"
98.59 270.92 0.01 0.02 "table"
88.49 270.70 0.17 0.46 "factor"
60.63 166.64 60.63 166.64 "matrix"
37.93 104.26 0.23 0.64 "sort"
37.62 103.40 37.62 103.40 "unique"
37.62 103.40 0.00 0.00 "inherits"
37.62 103.40 0.00 0.00 "as.factor"
0.39 1.08 0.00 0.00 "c"
```

19

20



Summary

- Fix bugs first.
- Examine system resource use.
- Look for bottlenecks.
- Examine processes that are resource hogs.
- Profile code running on real problem
- Identify small and expensive regions of code
- Tune ☺
- Repeat from the beginning

Cheat sheet for system usage:

1. Where's the bottleneck?

-Look at vmstat, if

- If CPU use (in vmstat) ~100%? => CPU bottleneck
- Else if heavy swapping => severe memory bottleneck
- Else if large # of page faults & context switches, small amount of memory free => moderate memory bottleneck
- Else if lots of processes in wait (see ps) and lot of i/o (see iostat, pmval, netstat) => I/O bottleneck

2. What's hogging resources?

- use ps, top or pmval/pmstat to watch system as it executes
- look for processes with large RSS, large CPU percentages, large #'s of major faults, or large # of I/o calls

3. Profile code:

- Use an appropriate tool for your programming language (hprof, dprof, grprof, rprof, strace)
- Run your program on real-life examples
- Look for small sections of code that consume lots of resources