Reprinted from the

Proceedings of the Linux Symposium

July 23rd–26th, 2008
Ottawa, Ontario
Canada

If I turn this knob... what happens?

Arnaldo Carvalho de Melo
Red Hat Inc.
`acme@{redhat.com,ghostprotocols.net}`

Abstract

Characterizing problems in systems with lots of configuration knobs while trying versions of components can be an error-prone task. System and application configuration details such as kernel boot options, SMP affinity, NIC and scheduler settings, `/proc` and `/sys` filesystem entries, `lock_stat` data, and other items can prove vital. Software to collect this information in a database, correlating to application performance numbers for automated and visual analysis, is needed to help in this process.

Work in this direction is presented in this paper, showing how changes in system tunings compare to previous results in the database. By automating the collection of performance numbers together with environment tunings, it helps in noticing trends in system behavior as system components evolve.

1 Introduction

Characterizing a performance or latency problem, benchmarking, testing new versions of system components or a new machine—all these require storing the results in a database or spreadsheet for comparisons. Creating graphics from the collected data also helps in this process.

The number of system (software and hardware) knobs keeps growing, and it is easy to overlook one setting and then have difficulty in reproducing it on another system with supposedly the same hardware and software components, as is common when trying to obtain help from fellow developers, a company help desk, or the support services of a vendor.

Automatically recreating a set of tunings after the update of one of the system hardware or software components and comparing the results of a series of benchmarks with previous results is important in the life cycle of any software.

A small variation in performance, latency, memory usage, and several other software metrics after the update of a component is usually acceptable. It is thus possible that continuous degradation of a metric is unnoticed over several development cycles as the base hardware used also gets upgraded.

This paper will describe efforts in providing software for reading and storing system settings in a database, independently from how these changes were performed, be it using basic system tools such as `chrt` or `taskset`, or using higher level tools, such as `tuna`, that will also be described.

Ways to do live analysis of changes on system components such as changing the scheduler policy, priority, and processor affinity of threads and interrupts will also be presented; as well as running benchmarks for post processing, storing results in a database, and then generating reports showing the sets of tunings that provided the best results; as well as graphics showing results from several sets of tunings.

2 Automated Testing

I started working in this area when trying to characterize performance degradations found when trying to run market data applications on the `PREEMPT_RT` Real Time enabled kernels.

Knobs such as enabling or disabling TSO (TCP Segmentation Offload), using private futexes by upgrading the system C library, disabling or enabling IRQ balancing, setting the affinity and priority of the hard IRQ threads, making sure that oprofile, systemtap or `lock_stat`[1] were not enabled, trying new patches by fellow developers, and many others quickly added up to make me crazy when trying to compare results.

Many times this leads to having to re-run tests already performed due to forgetting whether one of these many knobs had been set.

To help in comparing the results, I started working on a set of software components, mostly written in the Python language.

Some system interfaces lacked Python bindings, so one of the first tasks performed was to fill this gap.

Bindings for the interfaces exposed through the schedutils package were written, python-schedutils [3], allowing getting and setting the scheduler policy, real time priority, and processor affinity of threads.

Another Python binding, python-ethtool [4], allows getting and setting network interface drivers knobs such as TSO[1] and UFO;[2] these are hardware assists for common TCP and UDP operations that can greatly improve performance, but inherently can add delay as the network stack waits for more application buffers to coalesce into one big segment to send to the ethernet card in just one transfer. Disabling these features is one thing usually tried when characterizing a problem, as it involves software implementations both in the OS network stack and on the NIC firmware—two places where bugs can happen. Also, the interaction of these two software implementations can be a source of problems.

There is also a lot of information available in the `/proc` filesystem, that despite its initial goals of providing information about the processes in the system, has been overloaded with all sorts of system-wide information. A library that turns several areas of the information found there into Python dictionaries was also written.

Classes for turning `/proc/pid/stats` and status into dictionaries, for instance, are used by the other components to sample information such as the number of voluntary and involuntary context switches experienced by threads.

The sysctl information in `/proc/sys` is also turned into dictionaries and the ones that are changeable by the administrator (directly, using a simple `echo` shell command, or through higher level tools) are stored in a set of database tables, one per a selected set of `/proc/sys` subdirectories.

A tool that collects the state of this subset of sysctl settings and assigns a unique numeric identifier was also written.

[1]TCP Segmentation Offload

[2]UDP Fragmentation Offload

This tool initially was used to store more than just sysctl settings, with extra information such as if system analysis tools such as `lock_stat`, oprofile, or systemtap were in use, or the options passed through the kernel command line.

The tool is being rewritten so that it can be used just for sysctls, with another like-minded tool to be made available for other, non-sysctl knobs.

To better illustrate the use of such tool, here are some examples:

```
$ tuneit --show 1
tcp_congestion_control: bic
kcmd_idle: None
lock_stat: False
tso: lo=0,eth0=1
app_sched: SCHED_RR
app_affinity: ff
systemtap: False
vsyscall64: 1
kcmd_maxcpus: None
irqbalance: True
app_rtprio: 51
oprofile: False
$
```

This shows the set of tunings recorded in the database associated with the unique numeric identifier *1*.

To see what changed from the first set of tunings to the second one:

```
$ tuneit --diff 1,2
Tunings#: 1
  tso: lo=0,eth0=1
Tunings#: 2
  tso: lo=0,eth0=0
$
```

Only TSO was changed, being enabled on the first set of tunings for the `eth0` interface and disabled on the second set of tunings.

This tool can be used as well for identifying the sets of tunings where a knob had a particular value. For instance:

```
$ tuneit --query "vsyscall64=0"
71,111
```

Some work was done on allowing this tool to be used for replaying a specific set of tunings:

```
$ cat /proc/sys/kernel/vsyscall64
1
$ tuneit --replay 71
$ cat /proc/sys/kernel/vsyscall64
0
$
```

The settings that can not be replayed at run time, such as kernel command line options or the presence of kernel features such as lock_stat, will instead generate a warning, so that the user is aware when comparing the ensuing results.

Currently this suite also checks if lock_stat is enabled in the kernel, resetting it before running the benchmarks, and storing the contents of /proc/lock_stat immediately after its completion, so that later on they can be examined. The same procedure will be implemented for oprofile, when requested, so that one more of the best practices used by performance workers can be automated, avoiding cases where such valuable information gets lost, even having been collected.

Other tools in this suite are used to collect other relevant information such as details about the machine and the system components installed in it:

```
[root@doppio ait]# ./ait-get-sysinfo
arch: x86_64
cpu_model: Intel(R) Core(TM)2 CPU
         T7200  @ 2.00GHz
futex_performance_hack: None
irqbalance: False
kcmd_idle: None
kcmd_isolcpus: None
kcmd_maxcpus: None
kcmd_nohz: None
kernel_release: 2.6.25-rc8
lock_stat: False
nic_kthread_affinities: eth0=3
nic_kthread_rtprios:
nodename: doppio.ghostprotocols.net
nr_cpus: 2
oprofile: False
softirq_net_rx_prio:
softirq_net_tx_prio:
systemtap: False
tcp_congestion_control: bic
tcp_dsack: 1
tcp_sack: 1
tcp_window_scaling: 1
tso: lo=0,eth0=1,pan0=1
ufo: lo=0
vendor_id: GenuineIntel
vsyscall64: 0
[root@doppio ait]#
```

Existing tools such as sysreport will probably be used in the future, as they provide more information, although they take a considerably longer time to collect it.

Finally, to provide the information required for the reporting tools, the benchmark results are stored in the database, correlated with all the above information about the system hardware, software, and the respective settings of both.

One can report the best results in a textual form:

```
$ rit.py db perf4-1.lab.redhat.com \
            perf7-1.lab.redhat.com

server: perf4-1.lab.redhat.com

client: perf7-1.lab.redhat.com

latest report info:
  report id: 504
  kernel: 2.6.18-53.1.14.el5
  max rate: 25000

max rates per kernel release:

2.6.18-88.el5       : 100000
2.6.24-17.el5rt     : 97000
2.6.24-20.el5rt     : 100000
2.6.24-21.el5rt     : 94000
2.6.24-22.el5rt     : 64000
2.6.24.1-24.el5rt   : 100000
2.6.24.3-rt3.rwmult2: 100000
2.6.24.4-30nommapsem: 100000
2.6.24.4-41.el5rt   : 97000

rate: 1000
----------------------
Shared System tunings:

ufo: lo=0,eth0=0,eth1=0,eth2=0
softirq_net_tx_pri: 90,...,90
softirq_net_rx_pri: 90,...,90
app_rtprio: 0
irqbalance: False
app_affinity: ff
app_sched: SCHED_OTHER
kcmd_isolcpus: None
nic_kth_aff: eth1=ff;eth2=ff
nic_kth_rtpri: eth1=95;eth2=95
oprofile: False
systemtap: False
kcmd_maxcpus: None
futex_performance_hack: 0
kcmd_idle: poll
lock_stat: False
tcp_congestion_control: bic
client: perf7-1.lab.redhat.com
```

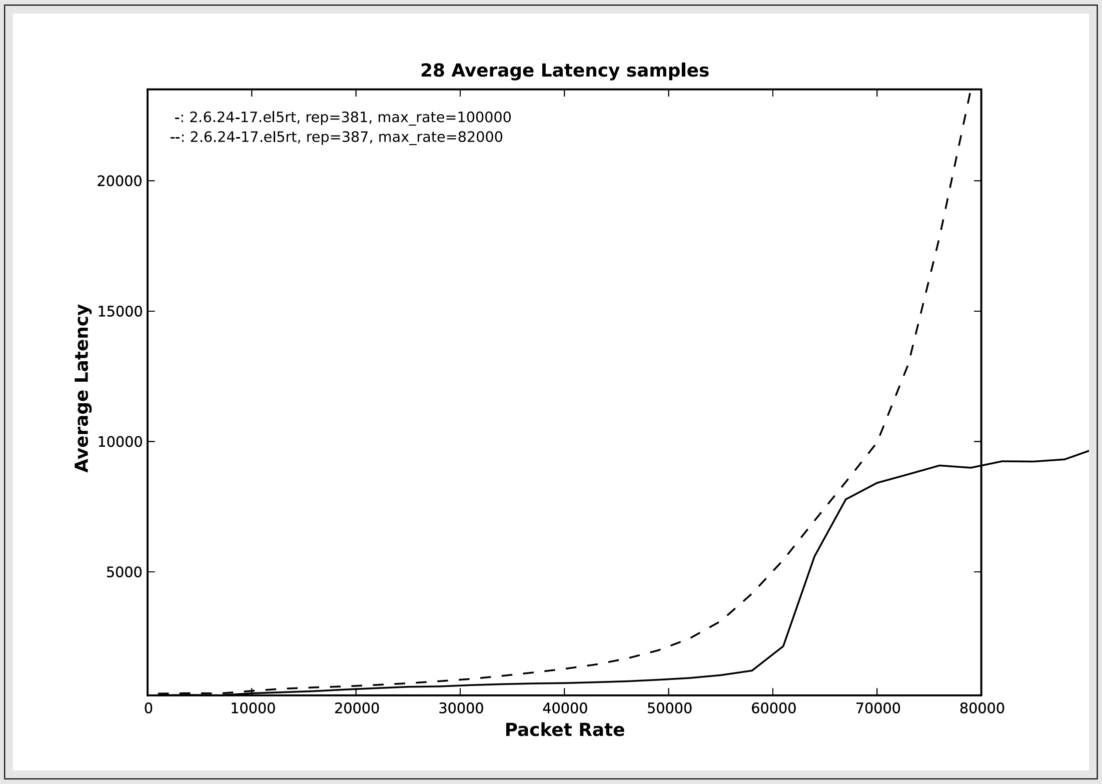

Figure 1: Solid line: libc-2.7.90 (uses private futexes), Dashed line: libc-2.5

```
Different system tunings:

env|tun|kernel    |avglat|   tso|vsc64|nhz
128|105| .24-22rt|249.38|eth2=1|   1 |  0
135|109| .24-20rt|252.41|eth2=1|   1 |off
127|104| .24-22rt|254.34|eth2=0|   1 |  0
136|109|24.1-24rt|258.53|eth2=1|   1 |off
134|108| .24-21rt|259.54|eth2=1|   1 |off
133|108|24.1-24rt|262.04|eth2=1|   1 |off
138|111|24.1-24rt|265.08|eth2=1|   0 |off
129|106| .24-22rt|266.51|eth2=1|   1 | on
130|106| .24-17rt|266.51|eth2=1|   1 | on
132|108| .24-17rt|267.97|eth2=1|   1 |off
```

The common set of tunings for all the 10 best test results is shown, followed by what really changed, and then the average latencies, the metric in this particular test.

Another result that can be generated is a set of graphs that show several benchmark results for visual comparison, as in Figure 1. This compares two versions of the system C library, one that uses private futexes and an older one that does not, on a system with 8 cores.

Another graphical report, this time with more than just two test results, is found in Figure 2, where the diamonds and squares lines are the ones with the old glibc, and all the tests have `lock_stat` on. The `html` file that includes these graphics has a link for the respective `/proc/lock_stat` data, where we can see that there is contention for `mmap_sem`, illustrated in Figure 3.

3 tuna

Also written was `tuna` [2], a tool to allow tweaking scheduler parameters, performing techniques such as CPU isolation.

It has three main boxes, one that shows the load for each CPU in the system, another with the interrupt sources, and the last one displaying the threads in the system.

Users can drag interrupt sources and threads into a CPU, setting the affinity of the dragged entities to that CPU.

Tuna also allows the user to right click on a CPU and isolate it, removing it from the CPU affinity masks of all threads and interrupt sources. It is also possible to do the opposite operation, including a CPU into the affinity masks of all interrupt sources and threads in the system.

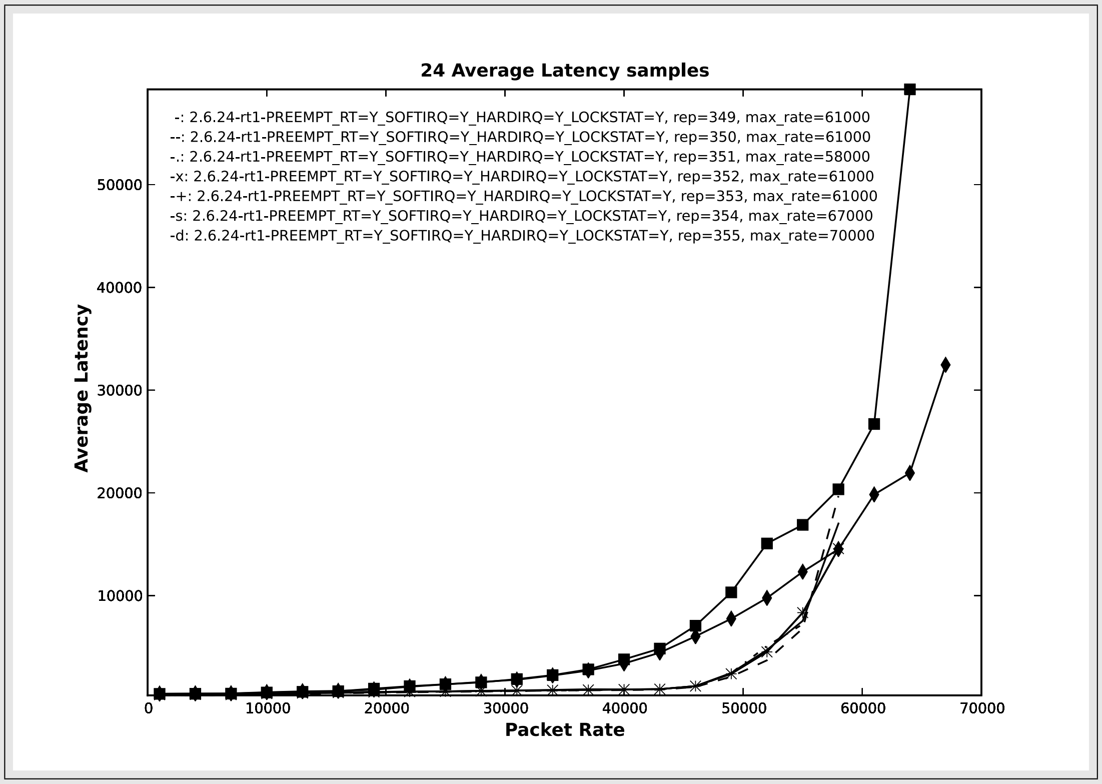

Figure 2: Diamonds and Squares: libc-2.5, Others: libc-2.7.90 (uses private futexes)

Usually the sequence is to isolate a CPU and then move some specific threads to that CPU so as to keep the isolated CPU cache hot, reducing accesses to main memory for a critical thread or set of threads.

More work is required to group cores per socket, for CPUs where caches are shared by the cores in a multi-core CPU socket. It then will be possible to move threads or interrupt sources to a socket and not just to a core.

The infrastructure put in place for multi-core CPUs will also allow creating groups that include not just cores in a socket, but any arbitrary grouping that is deemed useful for a particular purpose. That would permit arrangements such as dedicating two cores in a socket to a particular purpose, and the other two (assuming a quad-core CPU) for other purposes.

This will also help on big NUMA systems that have different costs for accessing memory that is one or more hops away from a particular core.

While it is understood that a general-purpose kernel tries hard to cope with all the underlying details on multi-core systems and NUMA topologies, it is generally considered useful to have such functionality for experimentation. As the complexity of such systems grows, a tool that exploits GUI facilities and provides higher level, simpler interfaces for performing these operations should be of help.

In trying to help a wider audience to understand more about the components in the kernel, `tuna` provides context-sensitive help, so far only for kernel threads. Right clicking on a kernel thread line in the threads box and clicking on the *What is This?* option opens up a window with information about it.

It is also possible to filter out kernel threads or user threads and sort by all the columns, and in the future it should be possible to add more columns from the fields found in the dictionaries built from the `/proc` files.

Most of the operations available in the GUI are also available on the command line, allowing its use in systems without graphical libraries.

```
lock_stat version 0.2
-------------------------------------------------------------------------------
    class name con-bounces contentions waittim-min  waittime-max waittime-total
-------------------------------------------------------------------------------

mm->mmap_sem-W     198626      265946         0.94      53724.80    17407373.03
mm->mmap_sem-R   21179643    49780404         0.74     299766.89  8739012694.54
--------------
  mm->mmap_sem   50039855 [<ffffffff802626ca>] rt_down_read+0xb/0xd
  mm->mmap_sem       4126 [<ffffffff80292c68>] sys_mprotect+0xce/0x22e
  mm->mmap_sem          0 [<ffffffff80211e7e>] sys_mmap+0xcf/0x119
  mm->mmap_sem          0 [<ffffffff80291014>] sys_munmap+0x32/0x59

...............................................................................

lock->wait_lock  39064428    40206507         0.94        316.86    56173839.20
---------------
lock->wait_lock   3584410 [<ffffffff804a1aed>] rt_spin_lock_slowunlock+0xf/0x5c
lock->wait_lock   8990657 [<ffffffff804a1bde>] rt_spin_lock_slowlock+0x21/0x19e
lock->wait_lock   3742935 [<ffffffff80262383>] rt_mutex_slowtrylock+0x18/0x79
lock->wait_lock    205398 [<ffffffff80262812>] rt_up_read+0x26/0x66

...............................................................................

dev->queue_lock#2 5918095     8546238         0.86      99690.79   506180455.19
-----------------
dev->queue_lock#2 1385877 [<ffffffff8043db2c>] __qdisc_run+0xa4/0x185
dev->queue_lock#2 7160361 [<ffffffff8042ddae>] dev_queue_xmit+0x12d/0x29f
dev->queue_lock#2       0 [<ffffffff8042d45d>] net_tx_action+0xbc/0xf3
```

Figure 3: Edited `lock_stat` data showing `mmap_sem` contention in report 354

4 Oscilloscope

The companion to `tuna` is an oscilloscope application. It should be fed with a stream of values that will be plotted on the screen together with a histogram.

The goal here is to be able to instantly see how a signal generator, such as `cyclictest`, `signaltest`, or even `ping`, reacts when (for instance) its scheduling policy or real time priority is changed, be it using `tuna` or plain `chrt`.

If the ftrace [5] feature is built into the running kernel, the oscilloscope classes will take snapshots of `/sys/kernel/debug/tracing/trace` and associate it with the position on the screen where the sample appears. So when the user clicks on the vicinity of such a sample, it will pop up a window with the ftrace-collected functions, usually what happened in the kernel while preemption and interrupts were disabled, causing the latency spike.

5 Future Directions

Integration with qpid [6] is planned, to get or set parameters on remote machines.

When this integration is complete, `tuna` will be just one of the interfaces to change knobs, another one being the AMQP Management Console, part of the qpid project [6].

Being able to use inventory systems for data about the systems used in the tests is also something to be considered.

The `tuneit` tool described earlier in this paper will be augmented to allow starting a tuning session. There it will look at all threads and interrupt sources in the system, recording the current scheduler policy, real time priority, and affinities. Then, after a `tuna`, plain `chrt` and `taskset`, or any other method, it will compare the new settings and record the changes in the database, allowing the settings to be replayed on the same machine or on another.

6 Conclusion

The activities that this paper describes should be familiar to many readers; ad-hoc ways to accomplish these goals probably have been performed by most.

I hope that by describing his efforts in this direction and talking about requirements for further usability encourages interested people to join forces with him in working on improving this infrastructure for wider use, saving work for people with similar needs.

Test results for more benchmarks (such as AMQP [6], netperf, and other open source benchmarks) will be performed and should be publicly available by July, in time for OLS 2008.

References

[1] lock stat documentation in the kernel sources `Documentation/lockstat.txt`

[2] tuna git repository `http://git.kernel.org/?p=linux/kernel/git/acme/tuna.git`

[3] python-schedutils git repository `http://git.kernel.org/?p=linux/kernel/git/acme/python-schedutils.git`

[4] python-ethtool git repository `http://git.kernel.org/?p=linux/kernel/git/acme/python-ethtool.git`

[5] ftrace tracing infrastructure `http://lwn.net/Articles/270971/`

[6] qpid project `http://cwiki.apache.org/qpid/`