Realtime Tuning 101

Tuning Applications on Red Hat MRG Realtime

Clark Williams

Agenda

- Day One
  - Terminology and Concepts
  - Realtime Linux differences from stock Linux
  - Tuning
  - Tools for Tuning
  - Tuning Tools Lab

Agenda

- Day Two
  - Measuring Realtime Performance
  - Causes of Excessive Latency
  - Locating Latency sources

What is *Real Time*?

- Real Time doesn't mean *Real Fast*
- Real Time implies:
  - Meeting Deadlines
  - Consistency
  - Low Latency

Deadlines

- Defined by application problem domain
- Usually two components to a deadline:
  - Period (may be random)
  - Maximum Latency
- The *period* is the interval at which events occur
- The *maximum latency* is the largest time delay acceptable in servicing the event before causing a *failure*

Consistency

- Standard Deviation is the measurement used here
- Being early sometimes as bad as being late
  - Think of a fuel injection system
- The goal is to be On Time
- Many people say *deterministic* when what they really mean is *consistent*

Low-Latency

- Latency is the time from the *occurrence* of an event to the time the event is *serviced*
- For example:
  - Event occurrence – timer interrupt
  - Service – user thread waiting for timer is scheduled and runs
- Latency must be small enough that it doesn't interfere with the *next* periodic event

How is Realtime Behavior Achieved?

- One way: Write everything yourself
  - *Supervisory Loop* programs
  - Written directly *on-the-metal*
- Our way: Use an OS with realtime features
  - At a minium:
    - Priority Hierarchy
    - Task Preemption

Priority Hierarchy

- Fixed priorities which tell the OS scheduler the relative importance of scheduled entities
  - Scheduling decision is “pick the highest priority”
- In Linux, there are multiple scheduling *policies*
  - SCHED_OTHER (also called SCHED_NORMAL)
  - SCHED_FIFO
  - SCHED_RR
  - SCHED_BATCH
- SCHED_FIFO and SCHED_RR are *realtime* policies

Linux Realtime Priorities

- SCHED_FIFO is basic realtime policy
  - SCHED_FIFO threads run until they block or exit
    - Realtime threads have no *quantum*
- SCHED_RR is modified SCHED_FIFO
  - Equal priority threads are round-robin scheduled using OS defined quantum
- Both share priority range of 1-99
  - SCHED_FIFO:1 is same priority as SCHED_RR:1
- Both realtime policies are scheduled before others on Linux

Preemption

- Preemption is a mechanism where a thread's execution is stopped at an arbitrary point and another thread is scheduled to run in its place
- On most OS's, hardware interrupts are the main points where preemption can occur
- Up until recently, the Linux kernel was only partially preemptable
  - Kernel threads were not preemptable if holding a lock
  - The -rt patchset changes this

Linux Realtime Timeline

- 2000 – 2005 – 2.4 kernel
  - CONFIG_PREEMPT
  - Low-latency patch
- 2005 – present – 2.6 kernel
  - CONFIG_PREEMPT_RT patch set
  - rt_mutex_t replaces most locks in kernel
    - Preemptable
    - Sleepable
  - Threaded interrupt handlers

Realtime Tuning

- Equal parts engineering and art
  - Requires knowledge of the deployment platform
  - Requires knowledge of the application architecture
  - Tries to adjust application behavior to take advantage of the platform
- Server tuning not always useful for RT applications
  - Things that help server throughput may actually hurt event response time

Too Many Overloaded Terms!

- Core
  - Processor
  - Snapshot of process memory after failure
- Socket
  - Network communications abstraction
  - Physical connection for processor package
- Thread
  - Schedulable entity in Linux kernel
  - Partial processor in Intel core

RT Terms

- Core – a processor
  - May be multiple cores in a physical package
- Thread – the schedulable entity on Linux
  - If I need to talk about multiple cpu threads on an Intel Xeon, I'll be explicit about it
- Socket
  - Mostly talking about physical processor socket
  - Will try to be explicit because we may talk networking sockets

Realtime Applications: Basic Building Blocks

- BIOS options
- System Services
- Cores/Sockets
- Threads
- Interrupts
- NUMA memory nodes
- Timers
  - High Resolution Timers (hrtimers)

BIOS Options

- Turn 'em off!
- Power Management
  - Intel SpeedStep
  - AMD PowerNow
- System Management Interrupts (SMI)
  - Not handled by OS, handled by BIOS code
  - Used by many system services
    - Thermal management
    - Error Detection and Correction
    - IPMI

System Services

- Cpuspeed – controls power management features
- Irqbalance – distributes IRQs among available cores based on load
- Usually best to turn both off and manually assign IRQs to cores

```
# /sbin/chkconfig cpuspeed off
# /sbin/chkconfig irqbalance off
```

Cores and Sockets

- A *core* executes machine instructions
- There may be multiple cores in a single physical *package* which is attached to a *socket*
- Cores have their own first level cache and share other levels of cache with other cores in a socket
- *Topology* is the description of how cores and sockets connected to memory and each other
  - /proc/cpuinfo contains topology information

Cores and Sockets (cont.)

- A core may be *isolated*
  - This means all non-critical threads are barred from executing on the core
  - Done currently using *thread affinity*
  - May be done later using *cpusets*
- On NUMA systems, cores are associated with a local memory node
  - For best realtime performance, try to avoid *cross-node* memory references (use libnuma calls)

Threads

- *Threads* are what the Linux scheduler manipulates
  - Mostly just a stack and a register context
- A *kernel thread* is a thread that runs in supervisor mode
- A *user thread* is a thread operating inside a user-mode virtual address space
- A *process* is the combination of address space, file descriptors, capabilities and one or more user threads

Threads (cont.)

- Threads have an *affinity* property which specifies the cores on which they can run
- sched_setaffinity(2) system call is the base mechanism for setting affinity
  - Both taskset(1) and tuna(8) use this internally
- You can use tuna to both isolate cores and set thread affinities:

```
# tuna –cpus=1 –isolate –threads='myapp*' –move
```

Interrupts

- Traditional interrupt handlers run in *interrupt context*
- MRG Realtime handlers run in a dedicated thread
  - This allows prioritization of interrupt handling
- Interrupts on Linux also have an *affinity* property to bind to a set of cores
  - /proc/irq/<n>/smp_affinity
- Setting interrupt smp_affinity also sets the RT IRQ thread affinity

Timers

- Original timer mechanism on Linux was jiffies based
  - Periodic timer interrupt at 100 or 1000 Hz
  - Best resolution was 10ms or 1ms
- High Resolution timers (hrtimers) added ability to have timer resolution to best that hardware can provide
  - LAPIC has 10ns resolution
  - HPET has 100ns resolution

Clocksources

- Hrtimers added clocksource abstraction
- A clocksource is something that provides read access to a monotonically increasing time value
  - Timestamp Counter Register (TSC)
  - High Performance Event Timer (HPET)
  - ACPI Power Management Timer (acpi_pm)
- Available clocksources found in the /sys filesystem under /sys/devices/system at:

  `clocksource/clocksource0/available_clocksources`

Sleeping

- Sleep and wakeup in defined increments
  - Nanosleep, usleep, clock_nanosleep
- Linux kernel internals use sched_timespec as which has seconds and nanoseconds fields.
- clock_nanosleep is the preferred timer mechanism
  - Uses struct timespec same as kernel
- Interval timers are traditional UNIX mechanism but signal handling code path is to complex to be completely deterministic

Timestamps

- What time is it right now?
- Two main calls
  - gettimeofday(2) – returns struct timeval
  - clock_gettime(2) – returns struct timespec
- clock_gettime marginally better since it uses struct timespec (so no conversion from internal Linux time)
- Both have VDSO entrypoints on 64-bit kernels
  - /proc/sys/kernel/vsyscall64=1 (default on rt)

Basic Tuning Goals

- Get the OS out of the applications way
  - Isolate cpu cores for application use
  - Set thread and IRQ affinity to isolated cores
  - Leave at least one core for general OS use
    - Safest to leave core0 for OS use
- Elevate the priority of important things
  - Make sure high priority threads block so that low priority threads can run
- Put related threads on same core/socket to maximize cache sharing

Tuning Tools

- Rtctl – service that sets thread priorities
  - Original use was to set IRQ thread priorities
- Tuna – set interrupt and thread affinities and priorities
  - GUI mode for exploration and tuning experiments
  - Command line mode for production
- Numactl – start applications with explicit NUMA policy rules

Kernel Boot Options

- idle=<type> - select idle loop behavior
  - mwait (default)
  - poll (best latency, highest power consumption
  - halt (deprecated in favor of mwait)
- nohz=on – turn on tickless behavior
  - Gets rid of periodic timer interrupts

rtctl

- Run at startup to set thread policy and priority
- Uses regular expressions defined in /etc/rtgroups
  - Format: <groupname>:<sched>:<prio>:affinity:<regex>
- <regex> field is used to match output of 'ps'
- Easily modified to set not just kthreads but application threads as well

tuna

- Split into gui and command line components
- Gui shows:
  - sockets/cores
  - interrupts
  - Threads
- GUI allows interactive tuning
- Command line used to isolate cpus and move threads/interrupts on demand
  - Typically used in startup script

numactl

- Control NUMA memory allocation policy when launching an application
- Allows you to bind an application to a group of cores or to a group of memory nodes
  - numactl –cpunodebind=0-4 – myapp --arg

Tuning Steps

- Turn off problematic BIOS options
  - Power management
  - SMI generators
    - Probably need to talk to the system vendor about these
- Turn off unneeded system services
- Launch the application and measure the performance
  - Baseline measurement with no tuning
- Now start tweaking

Tweaking?

- Figure out how to partition the application to use available cores and memory nodes
- Figure out priority hierarchy based on relative importance of application threads/processes
- Figure out what interrupts are important
  - To what cores should IRQs be bound?
  - What priority should the IRQ thread be?

Partitioning Strategy

- How many application processes/threads?
- How many available cores on system?
- What interrupts are important?
- Multiple NUMA nodes available?
- How much application memory needed?
- Threads/processes interacting?
  - Sharing buffers?
  - Communicating via pipes threads?

Partitioning Strategy (cont.)

- Always leave at least one core for the OS
  - Core zero is usually best
- Use tuna to determine topology, so you can group cores by socket
- Put threads that share data on same core or at least on same socket to maximize cache line sharing
-

Priority Hierarchy

- Important tasks should have higher priority
  - Make sure your definition of *important* matches the OS definition of important :)
- Insure that RT tasks don't hog the cpu
- Don't use priority 99!
  - Being one higher than another thread is as good as being 98 higher
- In general use the lowest priority possible
  - Try and stay below 50 to stay out of the way of the OS interrupts, softirqs, etc.

IRQs

- Boost important IRQ threads
  - Usually NIC interrupts more important than USB mouse interrupts
- Bind IRQs to the same core as application threads that consume data from the interrupt
  - Put threads receiving network data on the same core as NIC IRQ to get cache sharing benefit

Tuning Lab

- Apply tweaks to running cyclictest and watch performance change
- Setup AMQP latencytest between two systems and tune it (depending on available test systems)

Day Two
Advanced Tuning Roundtable

- 10G Ethernet
- RDMA
- NUMA
- SMIs