Kernel Performance Test on MontaVista Carrier Grade Edition 7 (CGE7)

MONTAVISTA SOFTWARE, LLC.

Prem Karat

Nov-2016

Contents

Abstract

This whitepaper captures list of performance tests done by MonvtaVista engineering team and the detailed steps on how the tests are executed on MontaVista Carrier Grade Edition 7 (CGE 7).

Overview of the following is covered in this paper

- Measurement of RT latency throught cyclictest.
- Measurement of various system latencies and bandwidth through LMbench.
- Measurement of network bandwidth through netperf.

Implementation Overview

Introduction

Unexplained delays while processing events are one of the biggest problems in the Realtime Linux kernel (or any realtime system for that matter). The term latency, when used in the context of the RT Kernel, is the time interval between the occurrence of an event and the time when that event is "handled" (typically "handled" means running some thread as a result of the event). Latencies that are of interest to kernel programmers (and application programmers) are:

- the time between when an interrupt occurs and the thread waiting for that interrupt is run
- the time between a timer expiration and the thread waiting for that timer to run
- the time between the receipt of a network packet and when the thread waiting for that packet runs

The timer and network example above are usually examples of the more general interrupt case (most timers signal expiration with an interrupt and most network interface cards signal packet arrival with an interrupt as well), but the main idea is that an "event" occurs and there is some elapsed time interval which concludes with the kernel successfully handling the event. So, latency in and of itself is not a bad thing; there is always a delay between occurrence and completion of an event. What is bad is when latency becomes excessive, meaning that the delay exceeds some arbitrary threshold. What is this threshold? That's for each application to define. A threshold or "deadline" is what defines a real time application: meeting deadlines means success, missing deadlines (exceeding the threshold) means failing to be real time.

Cyclictest

How do you measure event latency in the Linux kernel? Since there are many types of events that occur in the kernel, there's no one way to measure event latency. Usually a program is written to measure a specific type of latency. One such tool is called "cyclictest". Cyclictest measures the amount of time that passes between when a timer expires and when the thread which set the timer actually runs. It does this by taking a time snapshot just prior to waiting for a specific time interval (t1), then taking another time snapshot after the timer finishes (t2), then comparing the theoretical wakeup time with the actual wakeup time (t2 -(t1 + sleep_time)). This value is the latency for that timer wakeup. Cyclictest doesn't measure the entire event-to-response time, it measures the time from event to the start of real work (i.e. that which is influenced by the OS). The latency can come from: interrupts being disabled when the event occurs; other interrupts arriving; nested interrupt handling; wake-up of the cyclictest executable; wait until preemption is re-enabled or higher priority task yields or same-priority task is pre-empted; scheduler overhead; processor sleep states,

process migration, cache misses in the OS; lock priority inheritance etc.; and there may be interactions between these.

cyclictest gives you a single number for the total of all of this.

What cyclictest does is [3]

In one or more threads:

```
clock_gettime(&now)
while (not done)
    next = now + interval
    sleep(interval)
    clock_gettime(&now)
    latency = now – next
```

What Latency Includes

- Scheduler overhead
- IRQ overhead

Latency causes:

- IRQs disabled
- Preemption disabled
- IRQ handlers running in IRQ context
- Priority inversion
- Lock contention
- SMI
- Cache issues
- Higher priority threads
- etc..

To have better measurements, MV engineering team have designed to run different workloads together with cyclictest, and run cyclictest at a higher priority. The latency cyclictest sees in that situation is a good indication of the application's latency. You can also vary cyclictest's priority to see how the lower-priority threads of your application will behave. The different workloads that we run along with cyclictest include

- bonnie++ (file system benchmarking tool)
- netperf (generate tcp load on network)
- hackbench (Linux scheduler stress)

Test Execution:

All the 3 workloads explained above along with cyclictest are run using the parameters below for a user-specified time duration. (MV engineering team runs it for 15 mins)

Cyclictest execution parameters:

```
cyclictest -S -p 99  -m
```

where

-S: standard SMP testing: options -a -t -n and same priority of all threads
-m: lock current and future memory allocations
-p 99: priority of highest prio thread

Bonnie++ execution parameters

```
bonnie++ -u root -d <disk_backed_directory>
```

where

-d: scratch-dir (this directory should be on disk partition and not NFS)
-u: user

Hackbench execution parameters

```
hackbench -l 10000 -T -g 4
```

where

-l: number of loops
-T: threads (Each sender/receiver child will be a POSIX thread of the parent.)
-g: Defines how many groups of senders and receivers should be started.

Netperf execution parameters

```
netperf -H <netserver>
```

where

-H: host running netserver.
This will perform a TCP_STREAM test for 10 seconds. In MontaVista test environment, we run all the three workloads in parallel with cyclictest for 15 mins. An excerpt of the test script looks like this

Python Script Excerpt:

This will run the all the 4 scripts simultaneously for user-defined 'duration' in seconds (900 seconds)

```
benchmarks = [bonnie++.sh', cyclictest.sh',  hackbench.sh',
                netperf.sh']
for task in benchmarks:
        processes.append(Popen(task, preexec_fn=os.setpgrp))

time.sleep(duration)

for p in processes:
        os.kill(-p.pid, signal.SIGKILL)
```

The shell scripts are 'while true' bash scripts. For example the cyclictest.sh is:

```
#!/bin/bash

outfile=cyclictest.data

while true
do
        cyclictest -S -p 99 -m  >> $outfile 2>&1
done
```

The maximum latency value is then derived from cyclictest.data using the following filter commands.

```
cut -d ':' -f9 cyclictest.data | sed '/^$/d' | sed -e 's/^[ \t]*//' | sort -nru | head -1
```

LMbench [4]

lmbench is a series of micro benchmarks intended to measure basic operating system and hardware system metrics. LMbench helps in measuring bandwidth and latency. Most of the lmbench benchmarks use a standard timing harness and have a few standard options: parallelism, warmup, and repetitions. Parallelism specifies the number of benchmark processes to run in parallel. This is primarily useful when measuring the performance of SMP or distributed computers and can be used to evaluate the system's performance scalability. Warmup is the number of minimum number of microseconds the benchmark should execute the benchmarked capability before it begins measuring performance. Again this is primarily useful for SMP or distributed systems and it is intended to give the process scheduler time to "settle" and migrate processes to other processors. By measuring performance over various warmup periods, users may evaulate the scheduler's responsiveness. Repetitions is the number of measurements that the benchmark should take. This allows lmbench to provide greater or lesser statistical strength to the results it reports. The default number of repetitions is 11.

Test Execution:

The lmbench suite that we have used in our testing is lmbench-3.0-a9. Run the lmbench suite as per the README file in the lmbench src.

To run the benchmark, from lmbench source:
1. cd src
2. make results

While running make results, we provided the following configuration options.

MULTIPLE COPIES [default 1] <default>
Job placement selection: 1
MB <default>
SUBSET (ALL|HARWARE|OS|DEVELOPMENT) [default all] <default>
    # FASTMEM [default no] <default>
    # SLOWFS [default no] yes
    # DISKS [default none] none
    # REMOTE [default none] none
Processor mhz <default>
FSDIR [default /var/tmp] /tmp
Status output file [default /dev/tty] /dev/null
Mail results [default yes] no

it's a good idea to do several runs and compare the output. So

3. make rerun (repeat for 3-4 times)
4. make rerun
5. cd results; make

This should provide us with all the required parameters for our analysis. All the test results for various latencies and bandwidth will be present under *Results* folder. The following parameters are considered for monitoring system performance.

Simple syscall
Simple read
Simple write
Simple stat
Simple fstat
Simple open/close
Select on 10 fd's
Select on 100 fd's
Select on 250 fd's
Select on 500 fd's
Select on 10 tcp fd's
Select on 100 tcp fd's
Select on 250 tcp fd's
Select on 500 tcp fd's
Signal handler installation
Signal handler overhead
Protection fault
Pipe latency
AF_UNIX sock stream latency
Process fork+exit

Process fork+execve
Process fork+/bin/sh -c
integer bit
integer add
context switch

The list above is just a subset of parameters that we monitor. There are various parameters that could be monitored like TCP/UDP latency, Socket bandwith, Memory bzero bandwith, Memory load latency, etc. For a complete list of various latency and bandwith measurement/monitoring values, please refer to LMbench website/manual.

Netperf [5]

Netperf is a benchmark that can be used to measure various aspects of networking performance. Its primary focus is on bulk data transfer and request/response performance using either TCP or UDP and the Berkeley Sockets interface. The TCP stream performance test is the default test type for the netperf program. The simplest test is performed by entering the command:

```
/opt/netperf/netperf -H remotehost
```

which will perform a 10 second test between the local system and the system identified by remotehost. The socket buffers on either end will be sized according to the systems' default and all TCP options (e.g. TCP_NODELAY) will be at their default settings.

Test Exectution:

The netperf version used in our lab is 2.6.0. The test setup includes the following configuration. Since most of embedded platforms we support includes a NIC with a network speed of 1Gb/s, the following configuration applies.

- Server running netserver will have a NIC that supports 1Gb/s
- 1 Gb/s network switch that connects the server and the embedded target
- Embedded target (the node under test) will have NIC that supports 1 Gb/s

All network cards will on both server and client will support Full Duplex and auto-negotiation. Please use ethtool to confirm the settings above on both the server and the client.

Start netserver on the server using the following command

```
netserver &
```

On the target netperf is exectuted with following parameters

```
netperf -l 60 -H <netserver> -t TCP_STREAM -c 100 -C 100 -i 10,2 -I 99,5
-- -m <message size> -s  <local send & recv socket size> -S <remote send & rec
socket size>
```

where

-t : test name
-c, -C: Report local and remote cpu usage respectively
-i: the max and min number of iterations
-I: confidence level and confidence interval in percentage
-s, -S: local and remote send and receive socket sizes.
-m: Message size.

The entire tcp stream test is run with varying socket and message sizes.

The socket size used in testing is

```
socketSize = [128K, 57344, 32768, 8192]
```

The message size various from 10 bytes to 1G

```
messgSize = ['10', '100', '512', '1K', '16K', '256K', '512K', '1M',
             '16M', '256M',  '512M', '1G']
```

The testing involves sending each of those packet sizes for each socket size under test. So the test script looks like

Python Script Excerpt

```
for socket in socketSize
    for messg in messgSize:
        netperf -l 60 -H <netserver> -t TCP_STREAM -c 100 -C 100 -i 10,2 -I
99,5 -- -m messg -s  socket -S socket
```

The test results will capture the following network throughput

Socket 262142 Message 10
Socket 262142 Message 100
Socket 262142 Message 512
Socket 262142 Message 1024
Socket 262142 Message 16384
Socket 262142 Message 262144
Socket 262142 Message 524288
Socket 262142 Message 1048576
Socket 262142 Message 16777216
Socket 262142 Message 268435456
Socket 114688 Message 10
Socket 114688 Message 100
Socket 114688 Message 512
Socket 114688 Message 1024
Socket 114688 Message 16384
Socket 114688 Message 262144
Socket 114688 Message 524288
Socket 114688 Message 1048576
Socket 114688 Message 16777216
Socket 114688 Message 268435456
Socket 65536 Message 10

Socket 65536 Message 100
Socket 65536 Message 512
Socket 65536 Message 1024
Socket 65536 Message 16384
Socket 65536 Message 262144
Socket 65536 Message 524288
Socket 65536 Message 1048576
Socket 65536 Message 16777216
Socket 65536 Message 268435456
Socket 16384 Message 10
Socket 16384 Message 100
Socket 16384 Message 512
Socket 16384 Message 1024
Socket 16384 Message 16384
Socket 16384 Message 262144
Socket 16384 Message 524288
Socket 16384 Message 1048576
Socket 16384 Message 16777216
Socket 16384 Message 268435456

Appendices

[1] http://people.redhat.com/williams/latency-howto/rt-latency-howto.txt

[2]https://mindlinux.wordpress.com/2013/10/25/using-and-understanding-the-real-time-cyclictest-benchmark-frank-rowand-sony/

[3] https://events.linuxfoundation.org/slides/2011/linuxcon-japan/lcj2011_rowand.pdf

[4] http://lmbench.sourceforge.net/man/lmbench.8.html#toc

[5] http://www.netperf.org/netperf/training/Netperf.html


MontaVista