Using and Understanding the Real-Time Cyclictest Benchmark

Cyclictest results are the most frequently cited real-time Linux metric. The core concept of Cyclictest is very simple. However the test options are very extensive. The meaning of Cyclictest results appear simple but are actually quite complex. This talk will explore and explain the complexities of Cyclictest. At the end of the talk, the audience will understand how Cyclictest results describe the potential real-time performance of a system.

Frank Rowand, Sony Network Entertainment
February 20, 2013
What CyclicTest Measures

Latency of response to a stimulus.

- external interrupt triggers (clock expires)
  - possible delay until IRQs enabled
  - IRQ handling
  - cyclicTest is woken
  - possible delay until preemption enabled
  - possible delay until cyclicTest is highest priority
  - possible delay until other process is preempted
  - scheduler overhead

transfer control to cyclicTest
What Cyclic test Measures

Latency of response to a stimulus.

Causes of delay list on previous slide is simplified:

- order will vary
- may occur multiple times
- there are additional causes of delay
Many factors can increase latency

- additional external interrupts
- SMI
- processor emerging from sleep states
- cache migration of data used by woken process
- block on sleeping lock
  - lock owner gets priority boost
  - lock owner schedules
  - lock owner completes scheduled work
  - lock owner releases lock, loses priority boost
How Cyclictest Measures Latency

(Cyclictest Pseudocode)

The source code is nearly 3000 lines, but the algorithm is trivial
Test Loop

clock_gettime((&now))
next = now + par->interval

while (!shutdown) {
    clock_nanosleep((&next))
    clock_gettime((&now))
    diff = calcdiff(now, next)

    # update stat-> min, max, total latency, cycles
    # update the histogram data

    next += interval
}
The Magic of Simple

This trivial algorithm captures all of the factors that contribute to latency.

Mostly. Caveats will follow soon.
Cyclic test Program

main() {

    for (i = 0; i < num_threads; i++) {
        pthread_create((timerthread))
    }

    while (!shutdown) {
        for (i = 0; i < num_threads; i++)
            print_stat((stats[i]), i)
        usleep(10000)
    }

    if (histogram)
        print_hist(parameters, num_threads)
}
timerthread()

*timerthread(void *par) {

    # thread set up

    # test loop

}
Thread Set Up

stat = par->stats;
pthread_setsetaffinity_np((pthread_self()))
setscheduler({{par->policy, par->priority}})
sigprocmask((SIG_BLOCK))
Test Loop (as shown earlier)

clock_gettime(&now)
next = now + par->interval

while (!shutdown) {

clock_nanosleep(&next)

clock_gettime(&now)
diff = calcdiff(now, next)

# update stat-> min, max, avg, cycles
# Update the histogram

next += interval
}
Why show set up pseudocode?

The timer threads are not in lockstep from time zero.

Multiple threads will probably not directly impact each other.
The Magic of Simple

This trivial algorithm captures all of the factors that contribute to latency.

Mostly. Caveats, as promised.
Caveats

Measured maximum latency is a floor of the possible maximum latency

- Causes of delay may be partially completed when timer IRQ occurs

- Cyclic test wakeup is on a regular cadence, may miss delay sources that occur outside the cadence slots
Caveats

Does not measure the IRQ handling path of the real RT application

- timer IRQ handling typically fully in IRQ context
- normal interrupt source IRQ handling:
  - irq context, small handler, wakes IRQ thread
  - IRQ thread eventually executes, wakes RT process
Caveats

Cyclic test may not exercise latency paths that are triggered by the RT application, or even non-RT applications

- SMI to fixup instruction errata
- stop_machine()
- module load / unload
- hotplug
Solution 1

Do not use cyclic test.  :-)

Instrument the RT application to measure latency
Solution 2

Run the normal RT application and non-RT applications as the system load

Run cyclic test with a higher priority than the RT application to measure latency
Solution 2

Typical real time application will consist of multiple threads, with differing priorities and latency requirements.

To capture latencies of each of the threads, run separate tests, varying the cyclic test priority.
## Solution 2

### Example

<table>
<thead>
<tr>
<th>RT app thread</th>
<th>deadline constraint</th>
<th>latency constraint</th>
<th>RT app scheduler priority</th>
<th>cyclciest test priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>critical</td>
<td>80 usec</td>
<td>50</td>
<td>51</td>
</tr>
<tr>
<td>B</td>
<td>0.1% miss</td>
<td>100 usec</td>
<td>47</td>
<td>48</td>
</tr>
</tbody>
</table>
Cyclic test Latency
magenta: pri=51 blue: pri=48
Cyclic test Latency
magenta: pri=51 blue: pri=48
Aside:

Cyclic test output in these slides is edited to fit on the slides

Original:

$ cyclictest_0.85 -l1000000 -q -p80 -S

T: 0 (  460) P:80 I:1000 C:  100000 Min: 37 Act: 43 Avg: 45 Max: 68

Example of edit:

$ cyclictest_0.85 -l1000000 -q -p80 -S

T:0 I:1000 Min: 37 Avg: 45 Max: 68
T:1 I:1500 Min: 37 Avg: 42 Max: 72
Cyclictest Command Line Options

Do I really care???

Can I just run it with the default options???
Do I really care???

$ cyclictest_0.85 -l1000000 -q -p80
T:0  Min:  262  Avg:  281  Max:  337

$ cyclictest_0.85 -l1000000 -q -p80 -n
T:0  Min:  35  Avg:  43  Max:  68

-l1000000  stop after 1000000 loops
-q        quiet
-p80      priority 80, SCHED_FIFO
-n        use clock_nanosleep() instead of nanosleep()
red: clock_nanosleep()  blue: nanosleep()
thread: 0
red: clock_nanosleep()  blue: nanosleep()
thread: 0
Impact of Options

More examples

Be somewhat skeptical of maximum latencies due to the short test duration.

Examples are:

100,000 loops
1,000,000 loops

Arbitrary choice of loop count. Need large values to properly measure maximum latency!!!
## Priority of Real Time kernel threads for next two slides

<table>
<thead>
<tr>
<th>PID</th>
<th>PPID</th>
<th>S</th>
<th>RTPRIO</th>
<th>CLS</th>
<th>CMD</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2</td>
<td>S</td>
<td>1</td>
<td>FF</td>
<td>[ksoftirqd/0]</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>S</td>
<td>70</td>
<td>FF</td>
<td>[posixcputmr/0]</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>S</td>
<td>99</td>
<td>FF</td>
<td>[migration/0]</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>S</td>
<td>70</td>
<td>FF</td>
<td>[posixcputmr/1]</td>
</tr>
<tr>
<td>9</td>
<td>2</td>
<td>S</td>
<td>99</td>
<td>FF</td>
<td>[migration/1]</td>
</tr>
<tr>
<td>11</td>
<td>2</td>
<td>S</td>
<td>1</td>
<td>FF</td>
<td>[ksoftirqd/1]</td>
</tr>
<tr>
<td>353</td>
<td>2</td>
<td>S</td>
<td>50</td>
<td>FF</td>
<td>[irq/41-eth%d]</td>
</tr>
<tr>
<td>374</td>
<td>2</td>
<td>S</td>
<td>50</td>
<td>FF</td>
<td>[irq/46-mmci-pl1]</td>
</tr>
<tr>
<td>375</td>
<td>2</td>
<td>S</td>
<td>50</td>
<td>FF</td>
<td>[irq/47-mmci-pl1]</td>
</tr>
<tr>
<td>394</td>
<td>2</td>
<td>S</td>
<td>50</td>
<td>FF</td>
<td>[irq/36-uart-pl0]</td>
</tr>
<tr>
<td>Thread</td>
<td>Time</td>
<td>Min</td>
<td>Avg</td>
<td>Max</td>
<td>Command</td>
</tr>
<tr>
<td>------</td>
<td>-----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>---------</td>
</tr>
<tr>
<td>0</td>
<td>128</td>
<td>128</td>
<td>189</td>
<td>2699</td>
<td>T:0</td>
</tr>
<tr>
<td>0</td>
<td>125</td>
<td>125</td>
<td>140</td>
<td>472</td>
<td>T:0</td>
</tr>
<tr>
<td>0</td>
<td>262</td>
<td>262</td>
<td>281</td>
<td>337</td>
<td>T:0</td>
</tr>
<tr>
<td>0</td>
<td>88</td>
<td>88</td>
<td>96</td>
<td>200</td>
<td>T:0</td>
</tr>
<tr>
<td>0</td>
<td>246</td>
<td>246</td>
<td>320</td>
<td>496</td>
<td>T:0</td>
</tr>
<tr>
<td>1</td>
<td>253</td>
<td>246</td>
<td>315</td>
<td>509</td>
<td>T:0</td>
</tr>
<tr>
<td>0</td>
<td>35</td>
<td>35</td>
<td>43</td>
<td>68</td>
<td>T:0</td>
</tr>
<tr>
<td>0</td>
<td>34</td>
<td>34</td>
<td>44</td>
<td>71</td>
<td>T:0</td>
</tr>
<tr>
<td>0</td>
<td>38</td>
<td>38</td>
<td>43</td>
<td>119</td>
<td>T:0</td>
</tr>
<tr>
<td>0</td>
<td>36</td>
<td>36</td>
<td>43</td>
<td>65</td>
<td>T:0</td>
</tr>
<tr>
<td>1</td>
<td>37</td>
<td>36</td>
<td>45</td>
<td>78</td>
<td>T:1</td>
</tr>
<tr>
<td>0</td>
<td>36</td>
<td>36</td>
<td>44</td>
<td>91</td>
<td>T:0</td>
</tr>
<tr>
<td>1</td>
<td>37</td>
<td>37</td>
<td>45</td>
<td>111</td>
<td>T:1</td>
</tr>
<tr>
<td>0</td>
<td>34</td>
<td>34</td>
<td>44</td>
<td>94</td>
<td>T:0</td>
</tr>
<tr>
<td>1</td>
<td>34</td>
<td>34</td>
<td>43</td>
<td>104</td>
<td>T:1</td>
</tr>
</tbody>
</table>
T:0  Min: 123  Avg: 184  Max: 3814  live_update
T:0  Min: 125  Avg: 150  Max: 860   -q  no live_update
T:0  Min: 257  Avg: 281  Max: 371   -q -p80  SCHED_FIFO 80
T:0  Min:  84  Avg:  94  Max: 319   -q -n  clock_nanosleep
T:0  Min: 247  Avg: 314  Max: 682   -q -p80 -a -t  pinned
T:1  Min: 228  Avg: 321  Max: 506
T:0  Min:  38  Avg:  44  Max:  72   -q -p80 -n  SCHED_FIFO, c_n
T:0  Min:  33  Avg:  42  Max:  95   -q -p80 -a -n  pinned
T:0  Min:  36  Avg:  42  Max:  144   -q -p80 -a -n -m mem_locked
T:0  Min:  36  Avg:  44  Max:  84   -q -p80 -t -n  not_pinned
T:1  Min:  37  Avg:  45  Max:  94
T:0  Min:  36  Avg:  43  Max:  87   -q -p80 -a -t -n  pinned
T:1  Min:  36  Avg:  43  Max:  91
T:0  Min:  36  Avg:  43  Max:  141   -q -p80 -S  => -a -t -n
T:1  Min:  34  Avg:  42  Max:  88
blue: cpu pinned  red: cpu not pinned

thread: 0
blue: cpu pinned  red: cpu not pinned
thread: 1
Simple Demo  --  SCHED_NORMAL

- single thread

- clock_nanosleep(), one thread per cpu, pinned

- clock_nanosleep(), one thread per cpu

- clock_nanosleep(), one thread per cpu, memory locked

- clock_nanosleep(), one thread per cpu, memory locked, non-interactive
What Are Normal Results?

What should I expect the data to look like for my system?
Examples of Maximum Latency

#Platforms_Tested_and_in_Use_with_CONFIG_PREEMPT_RT

Platforms Tested and in Use with CONFIG_PREEMPT_RT

Comments sometimes include avg and max latency
table is usually stale

linux-rt-users email list archives

http://vger.kernel.org/vger-lists.html#linux-rt-users
Graphs of Maximum Latency

OSADL.org

Graphs for a wide variety of machines

List of test systems:

https://www.osadl.org/Individual-system-data.qa-farm-data.0.html
Full URL of previous graph


Typical command:

cyclicstest -l100000000 -m -Sp99 -i200 -h400 -q

OSADL Realtime QA Farm:

https://www.osadl.org/QA-Farm-Realtime.qa-farm-about.0.html

OSADL Latency plots:

https://www.osadl.org/Latency-plots.latency-plots.0.html
Additional OSADL Data

OSADL members have access to additional data, such as

- the data used to create the graphs
- the latency graphs extended in a third dimension, showing all test runs
Some Random Individual Systems

Picked from the OSADL spaghetti graph
Even “boring” graphs may contain interesting details.
Command Line Options

An unruly, out of control, set of control knobs
cyclictest V 0.85

Usage:
cyclictest <options>

-a [NUM] --affinity run thread #N on processor #N, if possible
    with NUM pin all threads to the processor NUM
-b USEC --breaktrace=USEC send break trace command when latency > USEC
-B --preemptirqs both preempt and irqsoff tracing (used with -b)
-c CLOCK --clock=CLOCK select clock
    0 = CLOCK_MONOTONIC (default)
    1 = CLOCK_REALTIME
-C --context context switch tracing (used with -b)
-d DIST --distance=DIST distance of thread intervals in us default=500
-D --duration=t specify a length for the test run
    default is in seconds, but 'm', 'h', or 'd' maybe added
    to modify value to minutes, hours or days
-e --latency=PM_QOS write PM_QOS to /dev/cpu_dma_latency
-E --event event tracing (used with -b)
-f --ftrace function trace (when -b is active)
-g MAX --of_MAX=MAX Report time in ms (up to MAX) for histogram overflows
-h --histogram=US dump a latency histogram to stdout after the run
    (with same priority about many threads)
-H --histofall=US same as -h except with an additional summary column
-i INTV --interval=INTV base interval of thread in us default=1000
-I --irqsoff Irqsoff tracing (used with -b)
-l LOOPS --loops=LOOPS number of loops: default=0(endless)
-m --mlockall lock current and future memory allocations
-M --refresh_on_max delay updating the screen until a new max latency is hit
-n --nanosleep use clock_nanosleep
-N --nsecs print results in ns instead of us (default us)
-o RED --oscope=RED oscilloscope mode, reduce verbose output by RED
-O TOPT --traceopt=TOPT trace option
-p PRIO --prio=PRIO priority of highest prio thread
-P --preemptoff Preempt off tracing (used with -b)
-q --quiet print only a summary on exit
-Q --priospread spread priority levels starting at specified value
-r --relative use relative timer instead of absolute
-R --resolution check clock resolution, calling clock_gettime() many
    times. list of clock_gettime() values will be
    reported with -X
-s --system use sys_nanosleep and sys_setitimer
-S --smp Standard SMP testing: options -a -t -n and
    same priority of all threads
-t [NUM] --threads=NUM number of threads:
    without NUM, threads = max_cpus
    without -t default = 1
-T TRACE --tracer=TRACER set tracing function
    configured tracers: blk function_graph wakeup_rt wakeup function nop
-u --unbuffered force unbuffered output for live processing
-U --numa Standard NUMA testing (similar to SMP option)
-v --verbose output values on stdout for statistics
    format: n:c:v n=tasknum c=count v=value in us
-w --wakeup task wakeup tracing (used with -b)
-W --wakeuprt rt task wakeup tracing (used with -b)
-X --dbg_cyclictest print info useful for debugging cyclictest
-y POLI --policy=POLI policy of realtime thread, POLI may be fifo(default) or rr
    format: -policy=fifo(default) or --policy=rr
Thread Behavior Options

- **-a [NUM] --affinity** run thread #N on processor #N, if possible with NUM pin all threads to the processor NUM
- **-c CLOCK --clock=CLOCK** select clock
  0 = CLOCK_MONOTONIC (default)
  1 = CLOCK_REALTIME
- **-d DIST --distance=DIST** distance of thread intervals in us default=500
- **-i INTV --interval=INTV** base interval of thread in us default=1000
- **-m --mlockall** lock current and future memory allocations
- **-n --nanosleep** use clock_nanosleep
- **-p PRIO --prio=PRIO** priority of highest prio thread
- **-Q --priospread** spread priority levels starting at specified value
- **-r --relative** use relative timer instead of absolute
- **-s --system** use sys_nanosleep and sys_setitimer
- **-S --smp** Standard SMP testing: options -a -t -n and same priority of all threads
- **-t --threads** one thread per available processor
- **-t [NUM] --threads=NUM** number of threads:
  without NUM, threads = max_cpus
  without -t default = 1
- **-U --numa** Standard NUMA testing (similar to SMP option) thread data structures allocated from local node
- **-y POLI --policy=POLI** policy of realtime thread, POLI may be fifo(default) or rr format: --policy=fifo(default) or --policy=rr

**side effect, sets -d0**

- **-h --histogram=US** dump a latency histogram to stdout after the run (with same priority about many threads)
  US is the max time to be be tracked in microseconds
- **-H --histofall=US** same as -h except with an additional summary column
Benchmark and System Options

-D --duration=t specify a length for the test run
default is in seconds, but 'm', 'h', or 'd' maybe added
to modify value to minutes, hours or days

-l LOOPS --loops=LOOPS number of loops: default=0(endless)

-e --latency=PM_QOS write PM_QOS to /dev/cpu_dma_latency
Display Options

- **g** MAX --of_max=MAX Report time in ms (up to MAX) for histogram overflows
- **h** --histogram=US dump a latency histogram to stdout after the run
  (with same priority about many threads)
  US is the max time to be be tracked in microseconds
- **-H** --histofall=US same as -h except with an additional summary column
- **-M** --refresh_on_max delay updating the screen until a new max latency is hit
- **-N** --nsecs print results in ns instead of us (default us)
- **-o** RED --oscope=RED oscilloscope mode, reduce verbose output by RED
- **-q** --quiet print only a summary on exit
- **-u** --unbuffered force unbuffered output for live processing
- **-v** --verbose output values on stdout for statistics
  format: n:c:v n=tasknum c=count v=value in us
Debug Options

-\texttt{b USEC} --breaktrace=USEC send break trace command when latency > USEC
-\texttt{-B} --preemptirqs both preempt and irqsoff tracing (used with -b)
-\texttt{-C} --context context switch tracing (used with -b)
-\texttt{-E} --event event tracing (used with -b)
-\texttt{-f} --ftrace function trace (when -b is active)
-\texttt{-I} --irqsoff Iqqsoff tracing (used with -b)
-\texttt{-O TOPT} --traceopt=TOPT trace option
-\texttt{-P} --preemptoff Preempt off tracing (used with -b)
-\texttt{-R} --resolution check clock resolution, calling clock_gettime() many times. list of clock_gettime() values will be reported with -X
-\texttt{-T TRACE} --tracer=TRACER set tracing function
 configured tracers: blk function_graph wakeup_rt wakeup function nop
-\texttt{-w} --wakeup task wakeup tracing (used with -b)
-\texttt{-W} --wakeuprt rt task wakeup tracing (used with -b)
-\texttt{-X} --dbg_cyclictest print info useful for debugging cyclic test
Debug Options

No time to describe in this talk

Hooks to invoke various tools that can capture the cause of large latencies
Options Trivia

Options parsing is not robust - example 1

# affinity will be 0
$ cyclictest -t -l1000 -a0
$ cyclictest -t -l1000 -a 0
$ cyclictest -t -l1000 -a7 -a0

# affinity will be 7, with no error message
$ cyclictest -t -l1000 -a7 -a 0

-a cpu affinity
Options Trivia

Options parsing is not robust - example 2

$ cyclicittest -ant
T: 0 (26978) P: 0 I:1000 C: 2091 Min:

$ cyclicittest -an -t
T: 0 (26980) P: 0 I:1000 C: 1928 Min:
T: 1 (26981) P: 0 I:1500 C: 1285 Min:

-a cpu affinity
-n clock_nanosleep()
-t one thread per cpu
Options Trivia

Options parsing is not robust

Best Practice:

- do not combine options
- specify each separately with a leading "-"
Third Data Format

Report each latency

```
$ cyclictest -q -n -t1 -p 48 -l 10000 -v
```
Cyclic test latency
pri=51 ping flood
Hitting the RT sched throttle

/proc/sys/kernel/sched_rt_runtime_us

/proc/sys/kernel/sched_rt_period_us

  cyclicittest:  SCHED_FIFO  priority=80

background load:
  - continuous
  - SCHED_FIFO  priority=40
sched_rt_runtime_us 80%
cycli test: SCHED_FIFO pri 80

latency (us ecc)

event number

5 x 10^4
4 x 10^4
3 x 10^4
2 x 10^4
1 x 10^4
0
sched_rt_runtime_us 80%
cycli
test: SCHED_FIFO pri 80

Latency (usec)

0 1000 2000 3000 4000

Event number
sched_rt_runtime_us-95_pct

cycliitest: SCHED_FIFO pri 80

latency (usec)

2.0 \times 10^4

1.5 \times 10^4

1.0 \times 10^4

5 \times 10^3

0

0

1000

2000

3000

4000

event number
sched_rt_runtime_us-95_pct
cylictest: SCHED_FIFO pri 80
Hitting the RT sched throttle

/proc/sys/kernel/sched_rt_runtime_us
/proc/sys/kernel/sched_rt_period_us

cyclicetest: SCHED_NORMAL

background load:
- continuous
- SCHED_FIFO priority=40
Hitting the RT sched throttle

Why is this measurement interesting???

Gives a picture of how much cpu is NOT used by the real time tasks
sched_rt_runtime_us 80%
cyclictst:  SCHED_NORMAL
sched_rt_runtime_us 80%
cyclic test: SCHED_NORMAL

Latency (usec) vs Event Number

- Y-axis: Latency (usec) with logarithmic scale from $10^1$ to $10^8$
- X-axis: Event Number from 0 to 4000

Graph shows the latency (in microseconds) plotted against the event number, with a logarithmic scale on the Y-axis, ranging from 10 to 100000000.
sched_rt_runtime_us-95_pct

cyclic test: SCHED_NORMAL

latency (usec)

0 1000 2000 3000 4000

event number
Demo - oscilloscope

cyclicittest_0.85 -t1 -n -p80 -i100 -o10 -v \ |
oscilloscope >/dev/null
oscilloscope screen shot
Fourth Data Format

Report time of each histogram overflow

Should be in next version of cyclictest (0.86?)

$ cyclictest -q -h 400 -g 1000

The same information can be extracted from the third data format (-v), but this method is lower overhead.
Finding and Building

git clone \
git://git.kernel.org/pub/scm/linux/kernel/git/clrkwllms/rt-tests.git

source: src/cyclictest/cyclictest.c

self-hosted:
  make

self-hosted without NUMA:
  make NUMA=0

cross-build without NUMA:
  make NUMA=0 CC="${CROSS_COMPILE}gcc"
Review

- Simple methodology captures all sources of latency fairly well
- Options must be used with care
- Options are powerful
- Different data formats are each useful
- Debug features can capture the cause of large latencies
THE END

Thank you for your attention...
Questions?
How to get a copy of the slides

1) leave a business card with me
2) frank.rowand@am.sony.com