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 Mobile Communications

October 25, 2013

131025_0328

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 Cyclictest 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.

Cyclictest 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_setaffinity_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.

September 2013 update

```
linux-rt-users
[rt-tests][PATCH] align thread wakeup times
Nicholas Mc Guire
2013-09-09 7:29:48
And replies
```

```
"This patch provides and additional
-A/--align flag to cyclictest to align
thread wakeup times of all threads
as closly defined as possible."

    "... we need both.

    same period + "random" start time
    same period + synced start time

    it makes a difference on some boxes
    that is significant."
```

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

- Cyclictest 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

Cyclictest 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 cyclictest. :-)

Instrument the RT application to measure latency

Solution 2

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

Run cyclictest with a higher priority than the RT application to measure latency

Solution 2

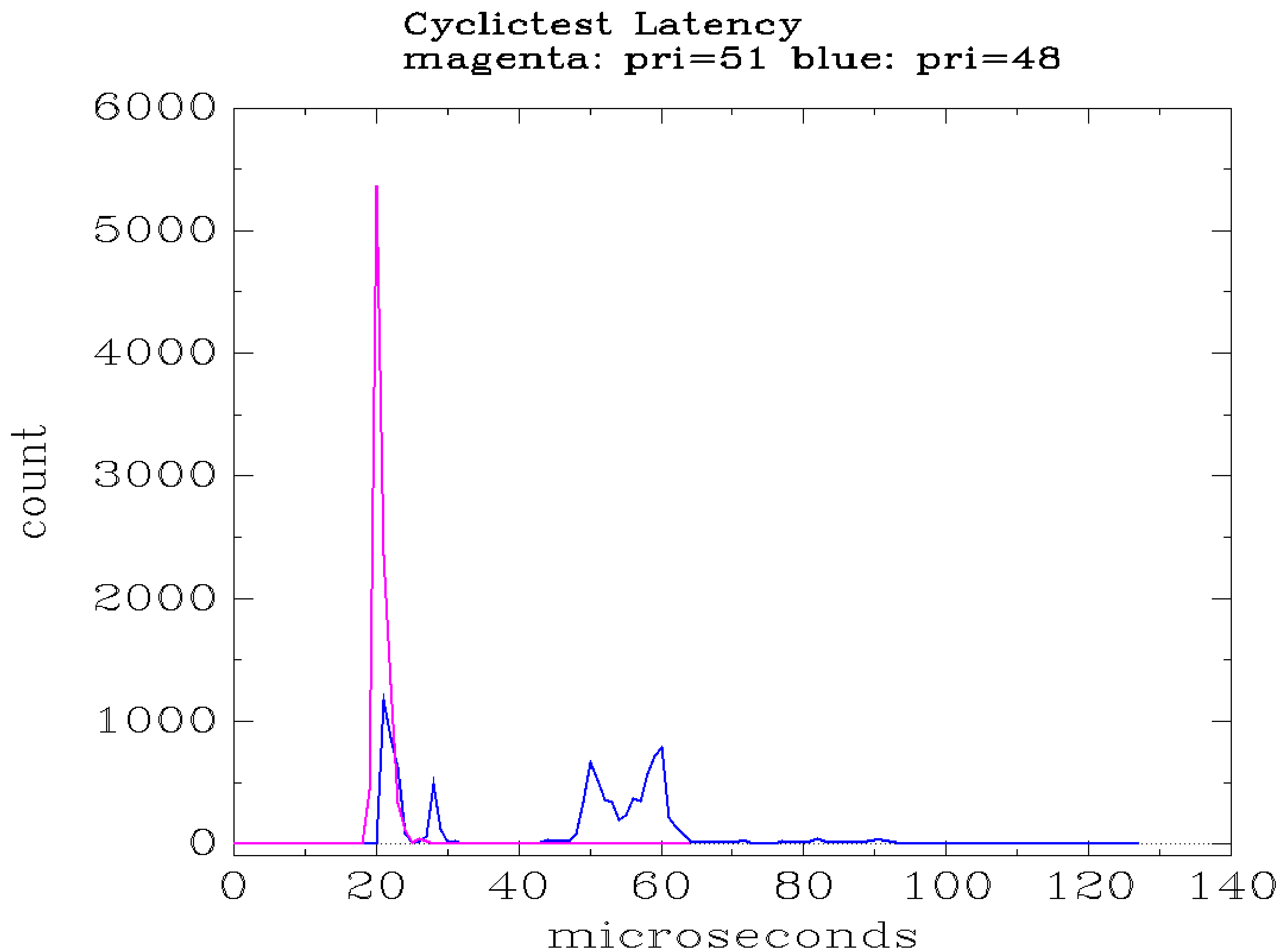
Typical real time application will consist of multiple threads, with differing priorities and latency requirements

To capture latencies of each of the threads, run separate tests, varying the cyclictest priority

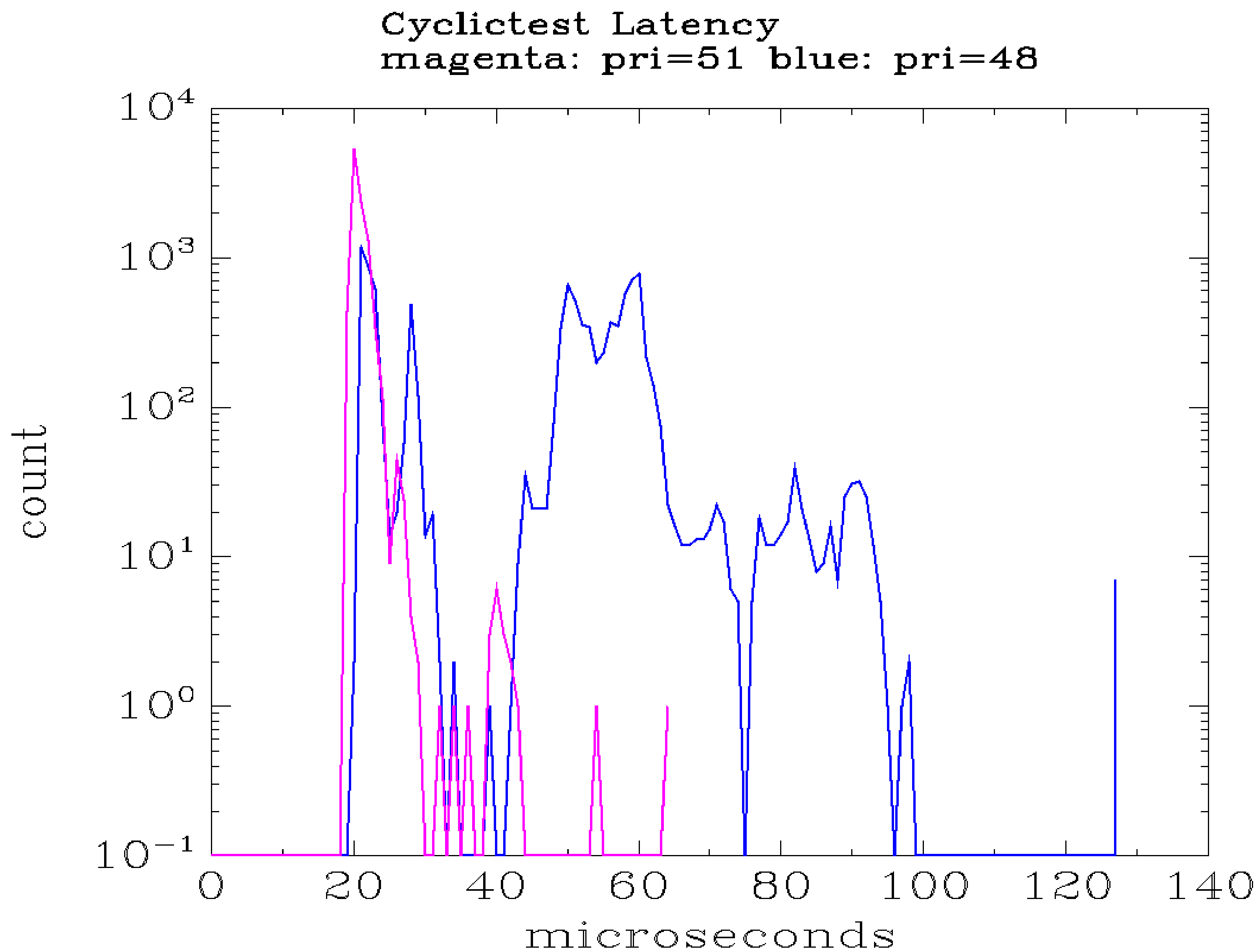
Solution 2

Example

RT app thread	deadline constraint	latency constraint	RT app scheduler priority	cyclictest priority
A	critical	80 usec	50	51
B	0.1% miss	100 usec	47	48



Cyclictest Latency
magenta: pri=51 blue: pri=48
6000
5000
4000
3000
2000
1000
0
count
0
20
40
60
80
100
120
140
microseconds


Cyclictest Latency
magenta: pri=51 blue: pri=48
count
microseconds
0
20
40
60
80
100
120
140
10^{-1}
10^{0}
10^{1}
10^{2}
10^{3}
10^{4}

Aside:

Cyclictest output in these slides is edited to fit on the slides

Original:

```
$ cyclictest_0.85 -l100000 -q -p80 -S

T: 0 (  460) P:80 I:1000 C: 100000 Min:     37 Act:   43 Avg:   45 Max:      68
T: 1 (  461) P:80 I:1500 C:  66675 Min:     37 Act:   49 Avg:   42 Max:      72
```

Example of edit:

```
$ cyclictest_0.85 -l100000 -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

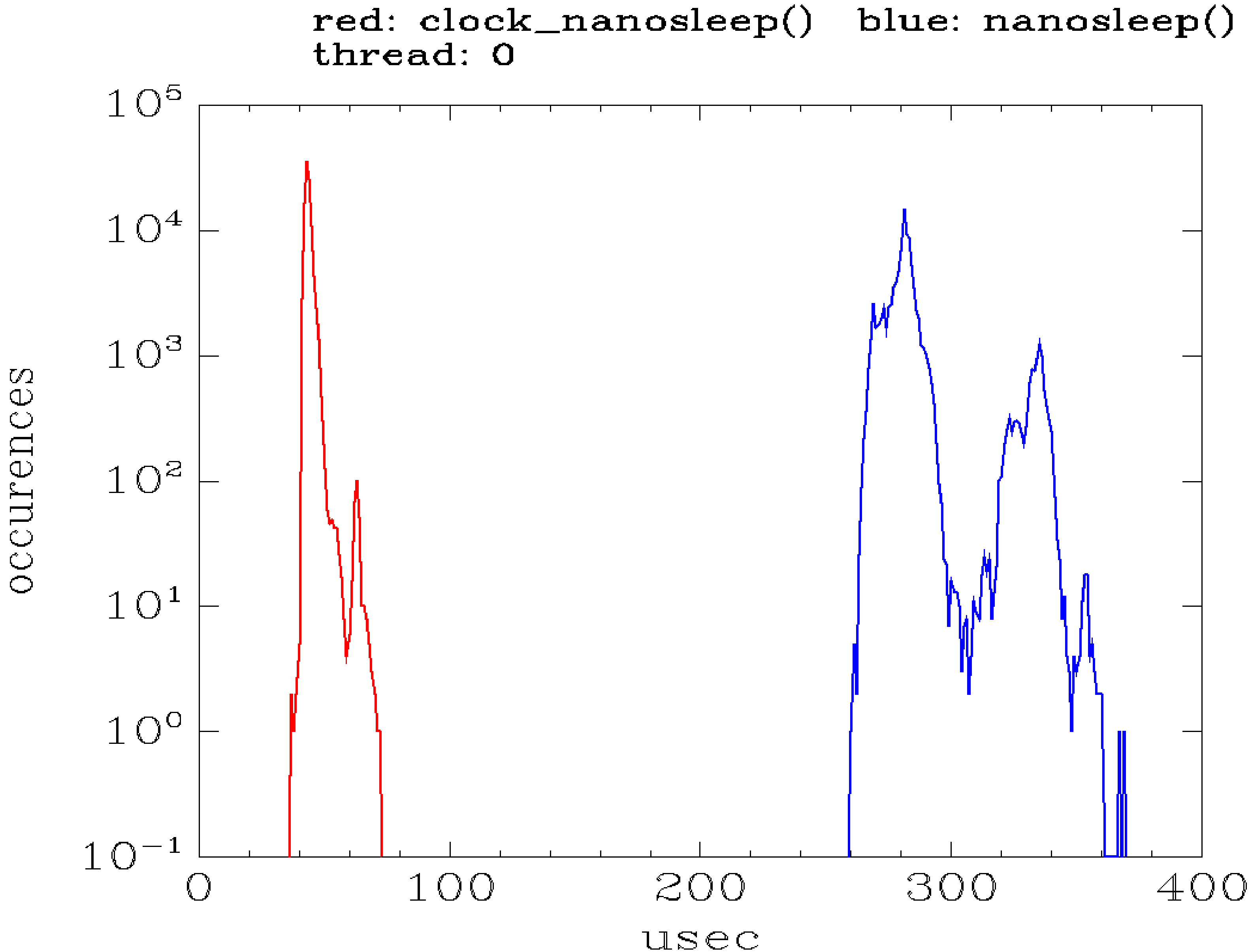
Do I really care???

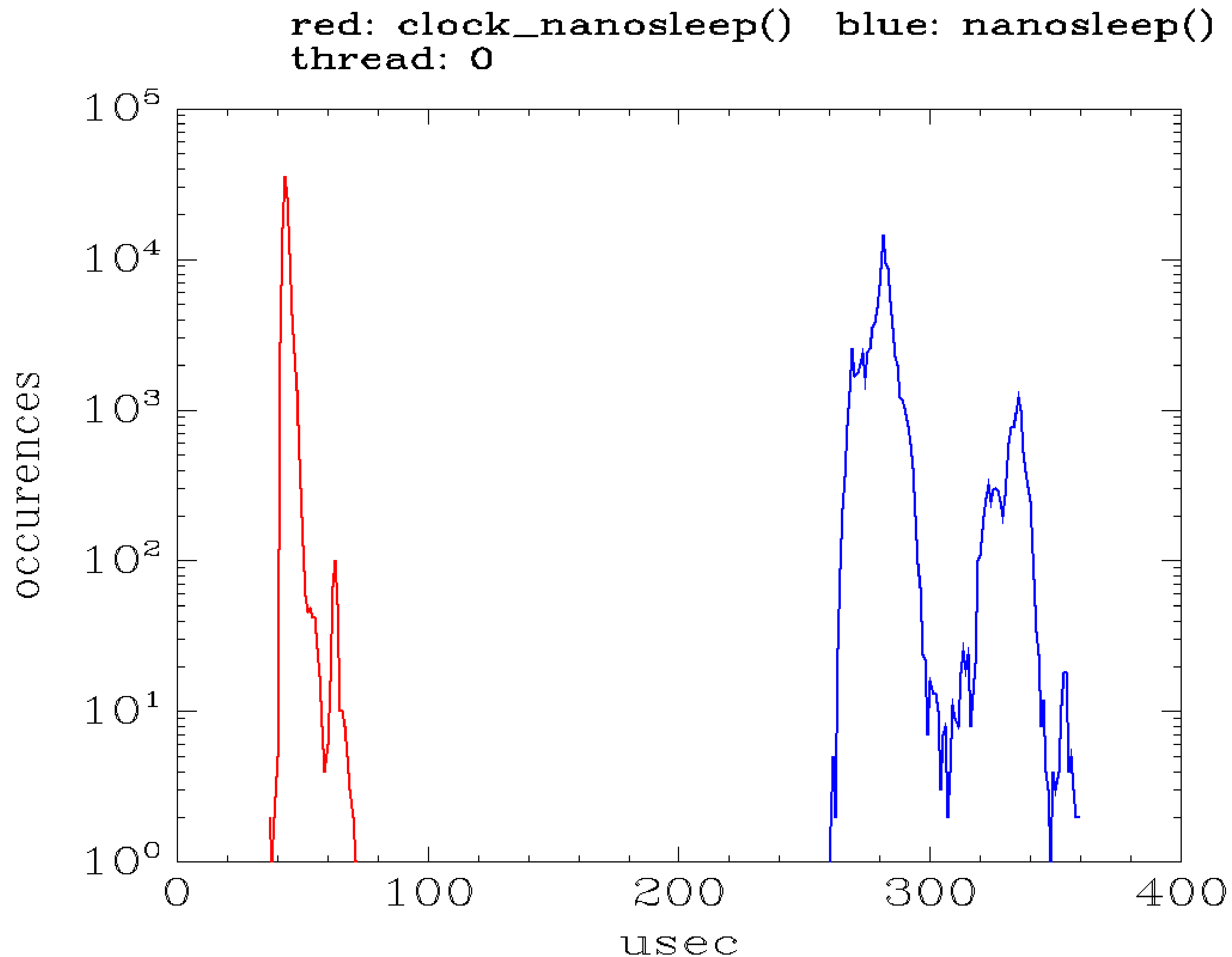
Can I just run it with the default options???

Do I really care???

```
$ cyclictest_0.85 -l100000 -q -p80
T:0  Min: 262  Avg: 281  Max:  337

$ cyclictest_0.85 -l100000 -q -p80 -n
T:0  Min:  35  Avg:  43  Max:   68
```

```
-l100000  stop after 100000 loops
-q        quiet
-p80      priority 80, SCHED_FIFO
-n        use clock_nanosleep()
          instead of nanosleep()
```


red: clock_nanosleep() blue: nanosleep()
thread: 0
occurences
usec
10^5
10^4
10^3
10^2
10^1
10^0
10^{-1}
0
100
200
300
400



red: clock_nanosleep() blue: nanosleep()
thread: 0
occurences
usec
10^0
10^1
10^2
10^3
10^4
10^5
0
100
200
300
400

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

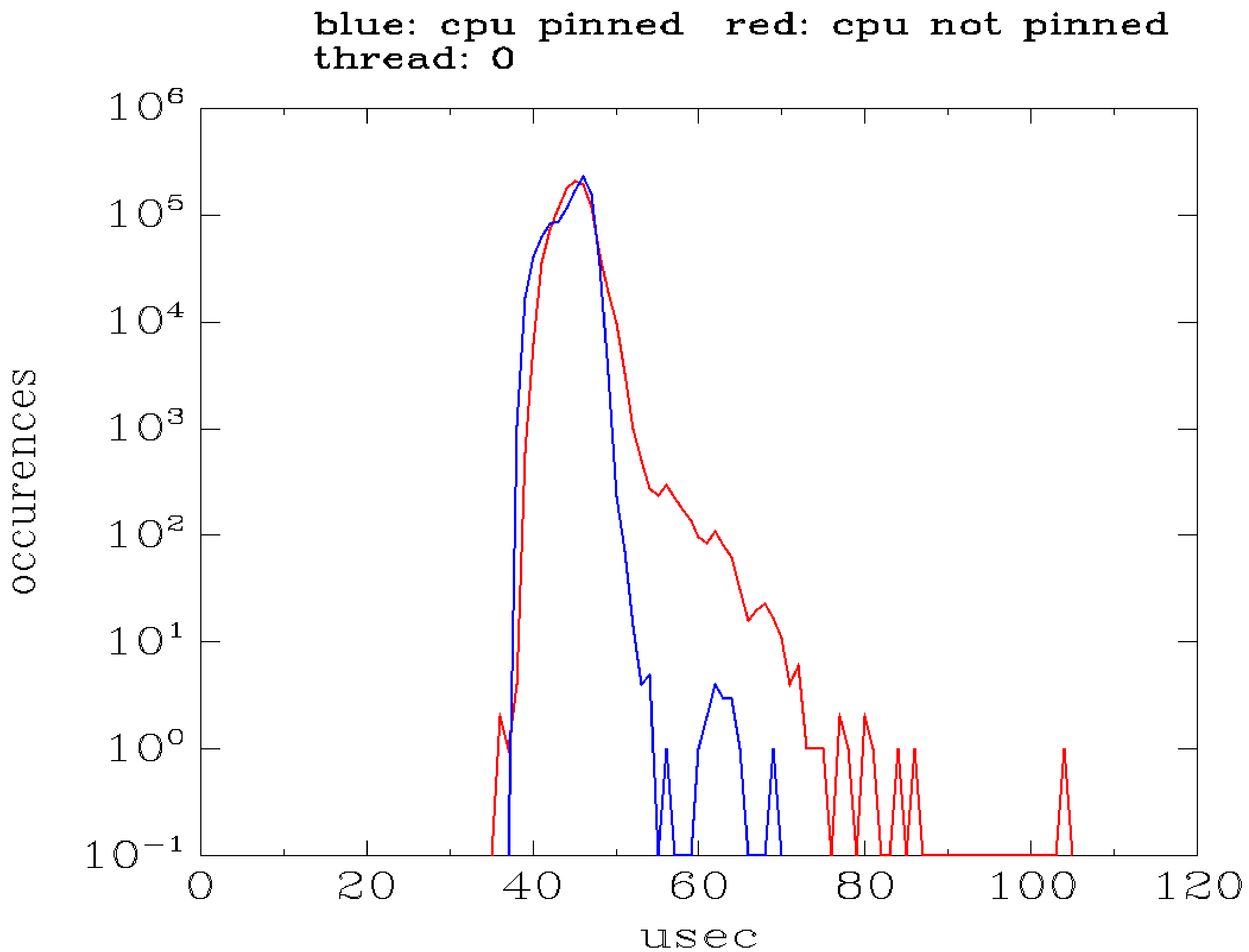
```
  PID  PPID S RTPRIO CLS CMD
    3     2 S      1 FF  [ksoftirqd/0]
    6     2 S     70 FF  [posixcputmr/0]
    7     2 S     99 FF  [migration/0]
    8     2 S     70 FF  [posixcputmr/1]
    9     2 S     99 FF  [migration/1]
   11     2 S      1 FF  [ksoftirqd/1]
  353     2 S     50 FF  [irq/41-eth%d]
  374     2 S     50 FF  [irq/46-mmci-pl1]
  375     2 S     50 FF  [irq/47-mmci-pl1]
  394     2 S     50 FF  [irq/36-uart-pl0]
```

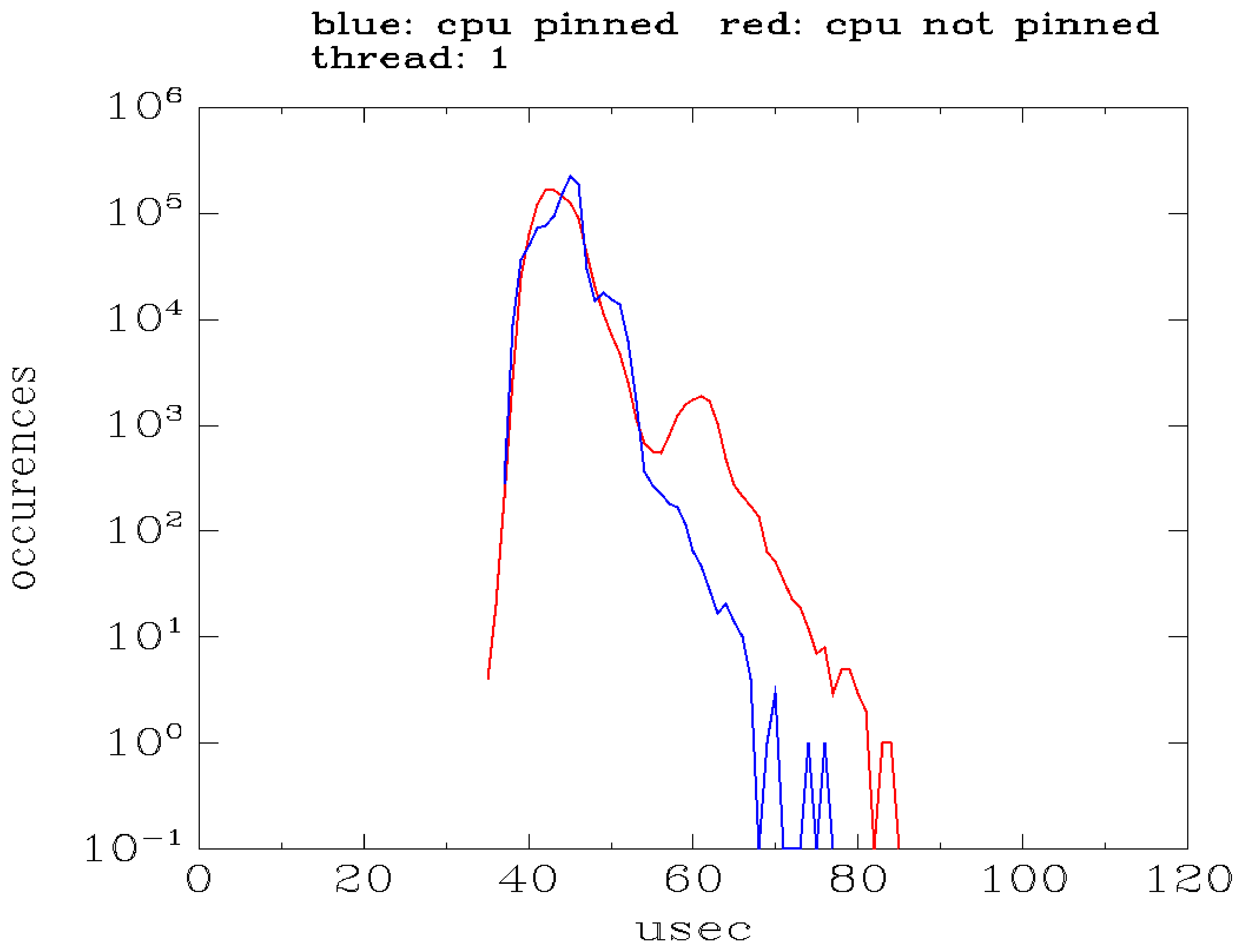
-l100000

```
T:0  Min: 128  Avg: 189  Max: 2699                      live update

T:0  Min: 125  Avg: 140  Max:  472   -q                 no live update

T:0  Min: 262  Avg: 281  Max:  337   -p80               SCHED_FIFO 80

T:0  Min:  88  Avg:  96  Max:  200   -n                 clock_nanosleep

T:0  Min: 246  Avg: 320  Max:  496   -q -p80 -a -t      pinned
T:1  Min: 253  Avg: 315  Max:  509

T:0  Min:  35  Avg:  43  Max:   68   -q -p80 -n         SCHED_FIFO, c_n

T:0  Min:  34  Avg:  44  Max:   71   -q -p80 -a -n      pinned

T:0  Min:  38  Avg:  43  Max:  119   -q -p80 -a -n -m mem locked

T:0  Min:  36  Avg:  43  Max:   65   -q -p80 -t -n      not pinned
T:1  Min:  37  Avg:  45  Max:   78

T:0  Min:  36  Avg:  44  Max:   91   -q -p80 -a -t -n pinned
T:1  Min:  37  Avg:  45  Max:  111

T:0  Min:  34  Avg:  44  Max:   94   -q -p80 -S         => -a -t -n
T:1  Min:  34  Avg:  43  Max:  104
```

-l1000000

```
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
occurences
usec
10^{6}
10^{5}
10^{4}
10^{3}
10^{2}
10^{1}
10^{0}
10^{-1}
0
20
40
60
80
100
120


blue: cpu pinned red: cpu not pinned
thread: 1
occurences
usec
10^6
10^5
10^4
10^3
10^2
10^1
10^0
10^{-1}
0
20
40
60
80
100
120

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

https://rt.wiki.kernel.org/index.php/CONFIG_PREEMPT_RT_Patch
#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

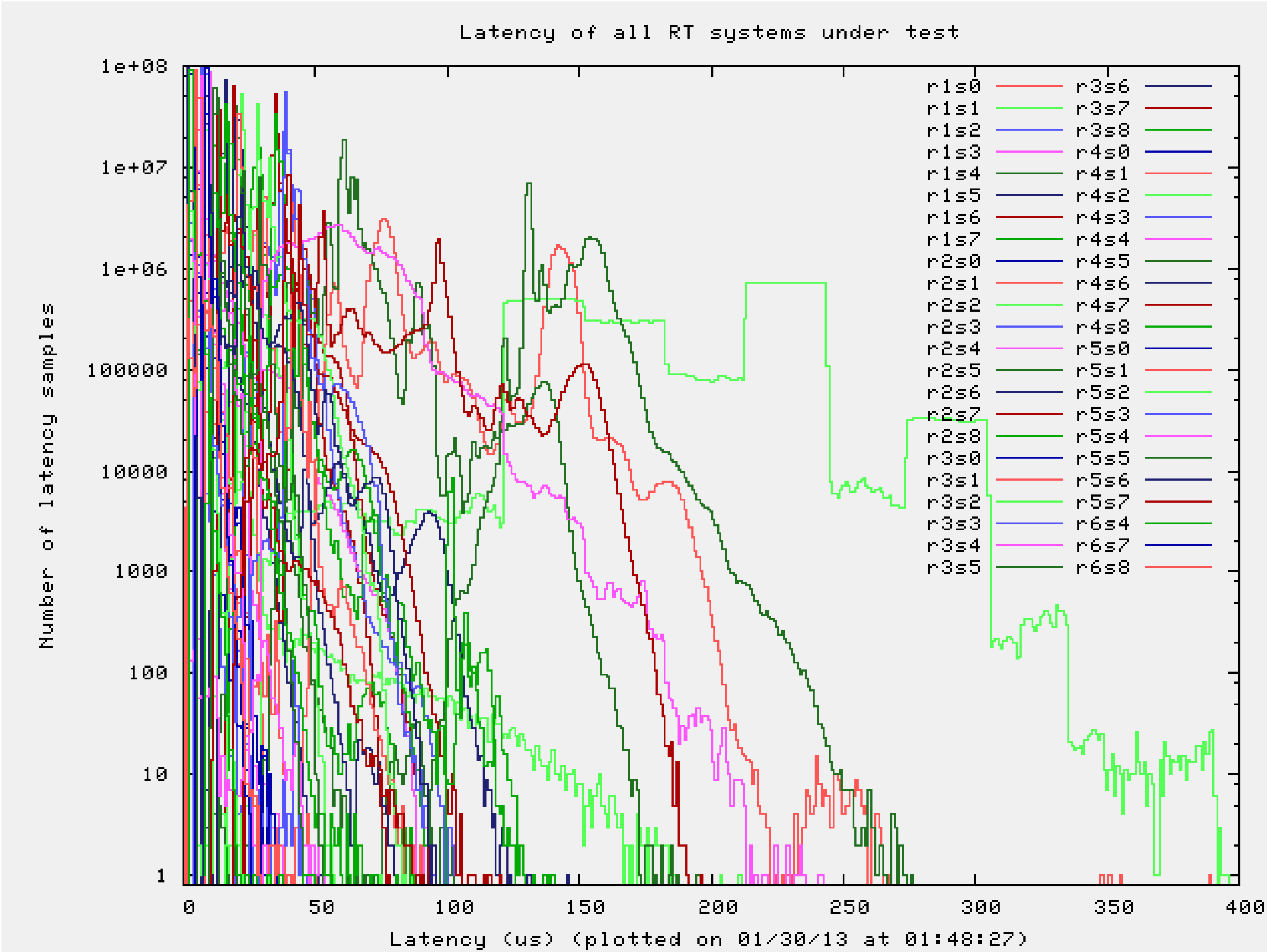
linux-rt-users email list archives

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



Latency of all RT systems under test
Number of latency samples
1e+08
1e+07
1e+06
100000
10000
1000
100
10
1
0
50
100
150
200
250
300
350
400
Latency (us) (plotted on 01/30/13 at 01:48:27)
r1s0
r1s1
r1s2
r1s3
r1s4
r1s5
r1s6
r1s7
r2s0
r2s1
r2s2
r2s3
r2s4
r2s5
r2s6
r2s7
r2s8
r3s0
r3s1
r3s2
r3s3
r3s4
r3s5
r3s6
r3s7
r3s8
r4s0
r4s1
r4s2
r4s3
r4s4
r4s5
r4s6
r4s7
r4s8
r5s0
r5s1
r5s2
r5s3
r5s4
r5s5
r5s6
r5s7
r6s4
r6s7
r6s8

Full URL of previous graph

https://www.osadl.org/Combined-latency-plot-of-all-RT-systems.qa-latencyplot-allrt.0.html?latencies=&showno=

Typical command:

cyclictest -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

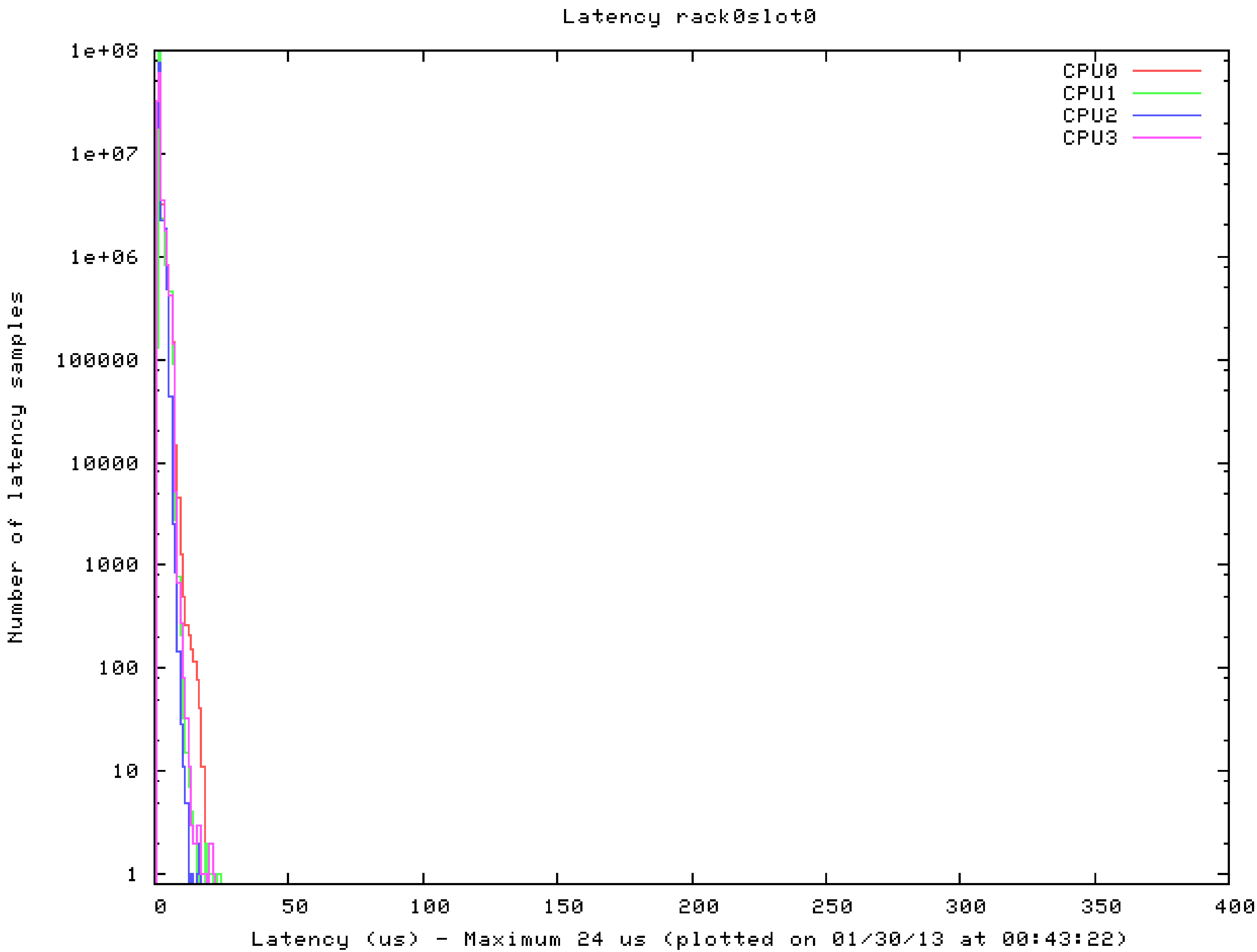
Additional OSADL Data

OSADL members have access to additional data, such as

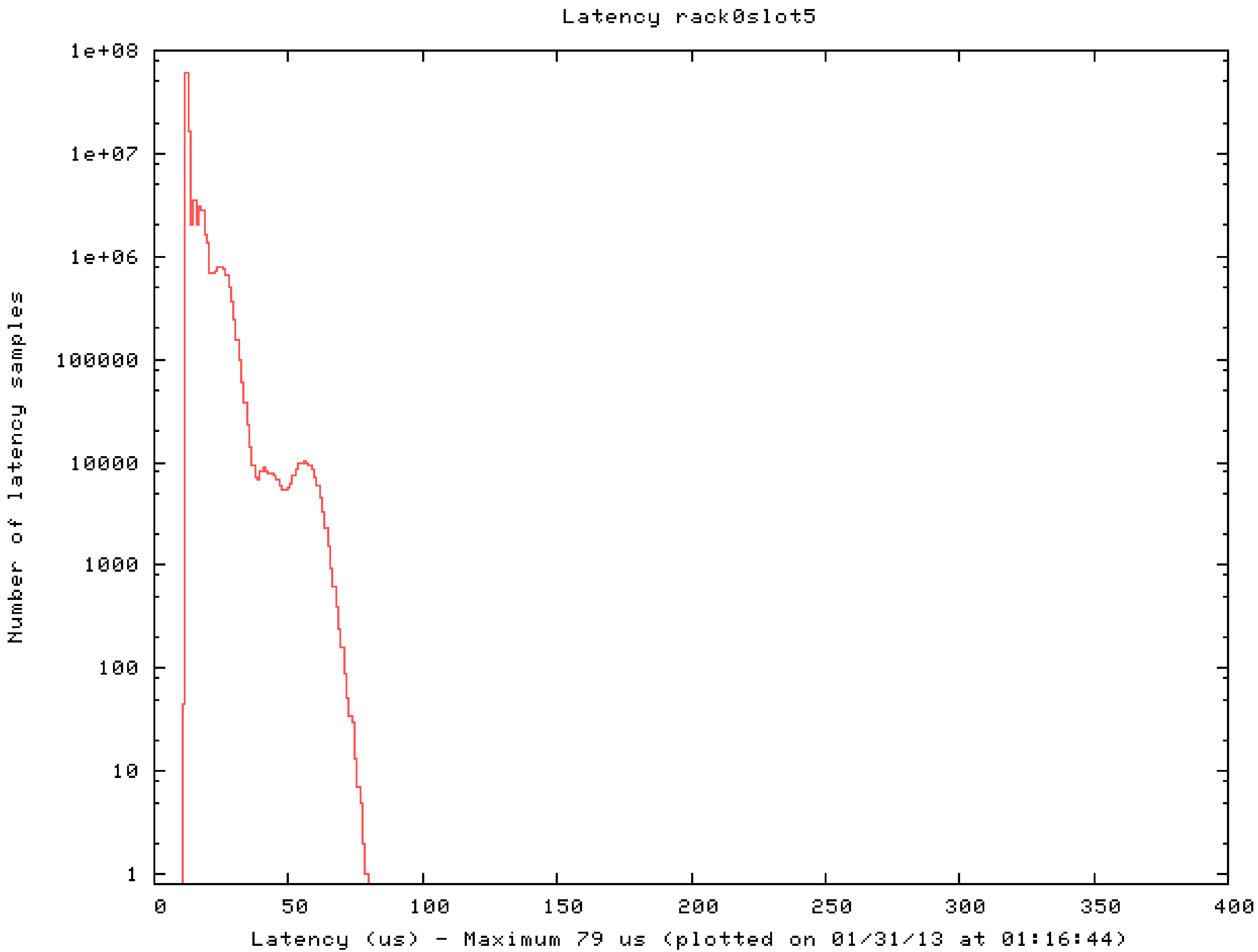
- the data used to create the graphs
- the latency graphs extended in a third dimension, showing all test runs

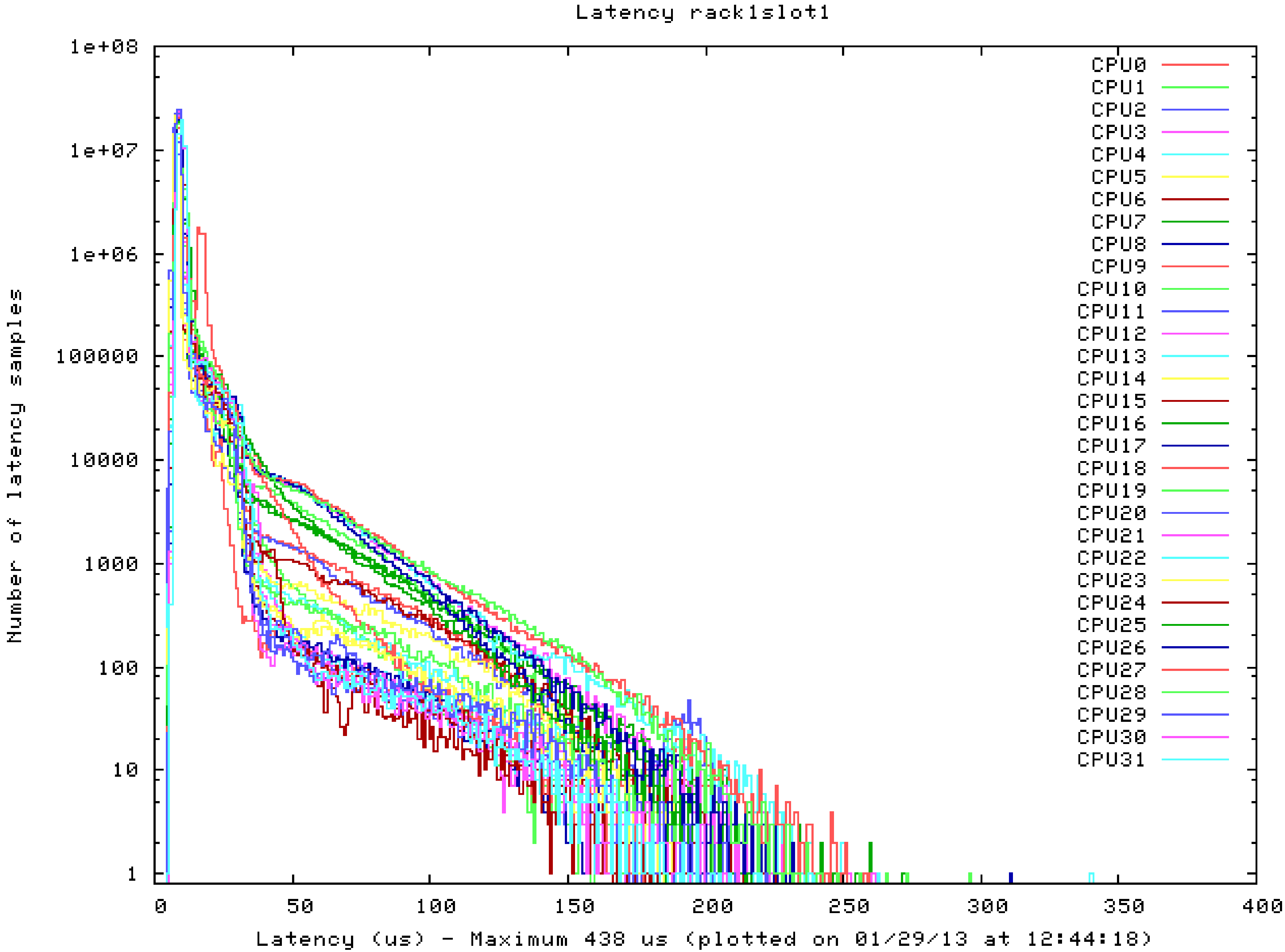
Some Random Individual Systems

Picked from the OSADL spaghetti graph

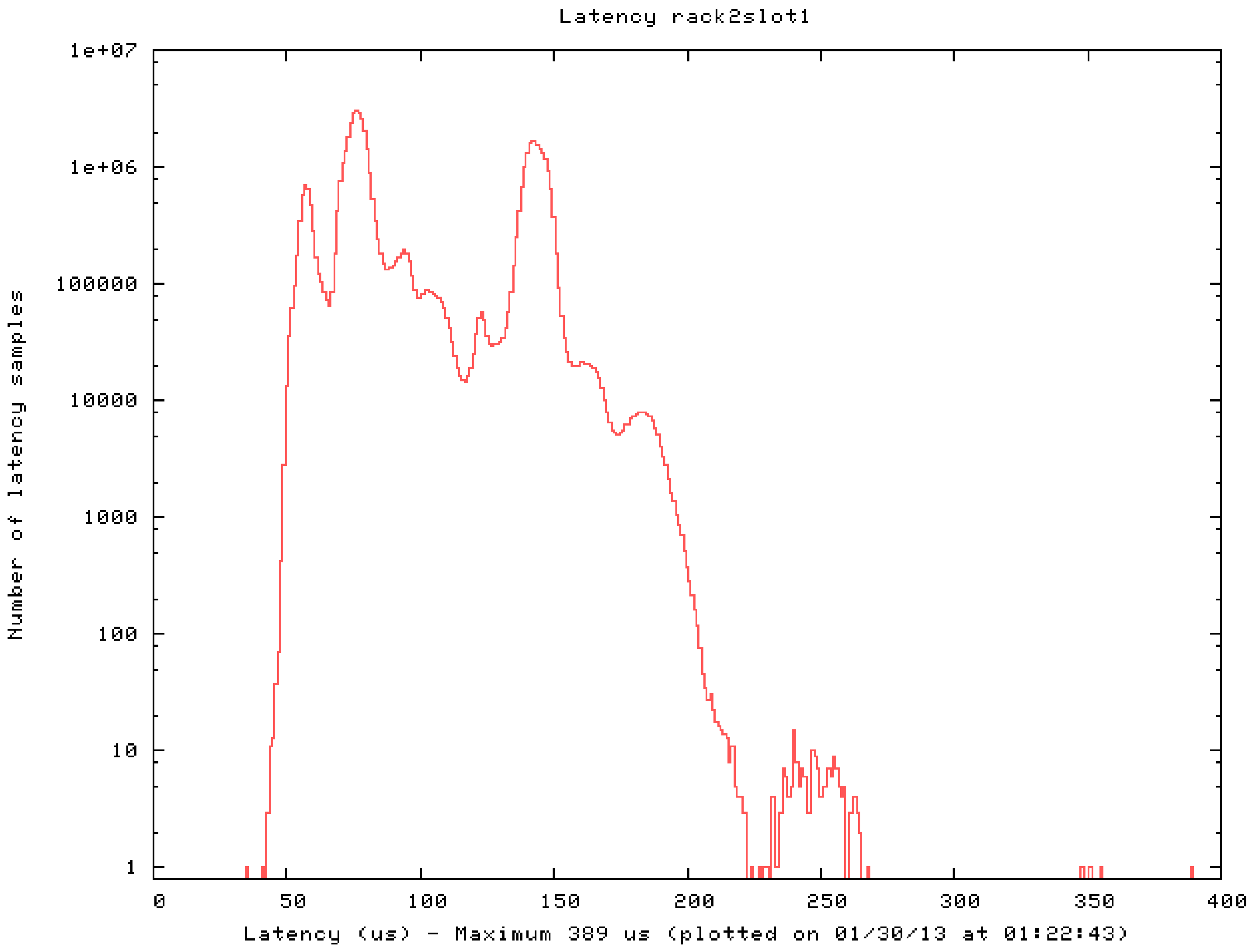


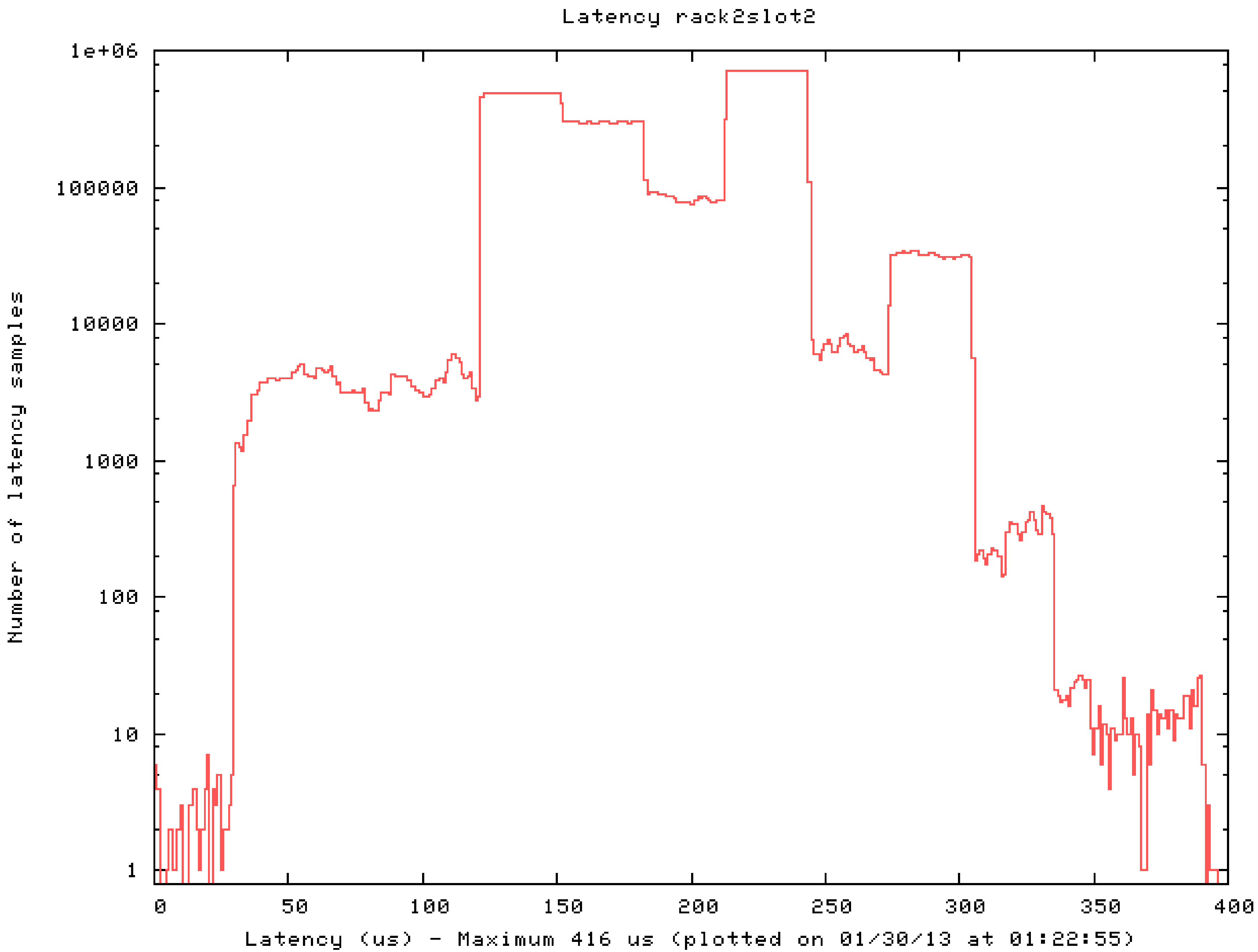
Latency rack0slot0
CPU0
CPU1
CPU2
CPU3
1e+08
1e+07
1e+06
100000
10000
1000
100
10
1
Number of latency samples
0
50
100
150
200
250
300
350
400
Latency (us) - Maximum 24 us (plotted on 01/30/13 at 00:43:22)

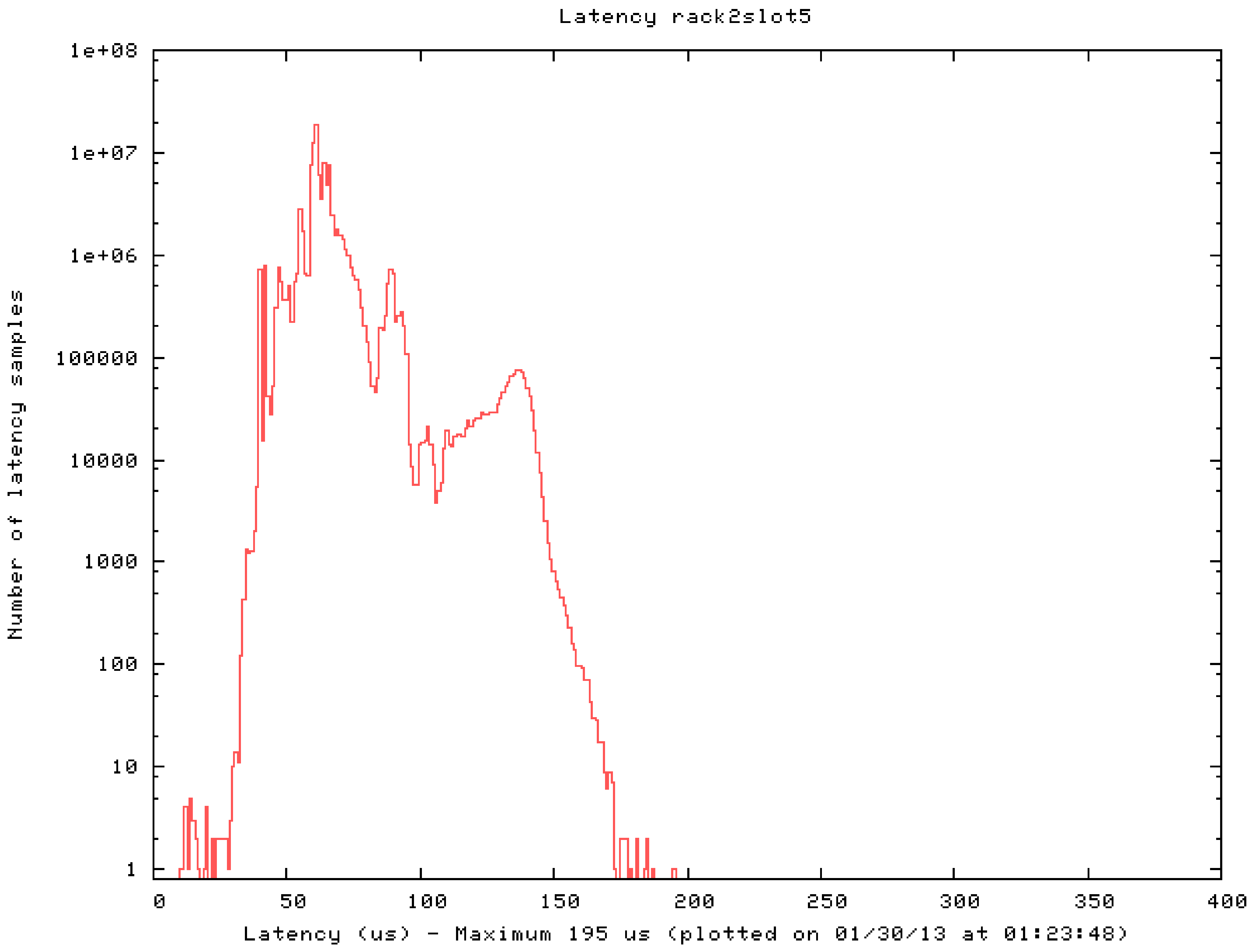

Latency rack0slot5
Number of latency samples
1e+08
1e+07
1e+06
100000
10000
1000
100
10
1
0
50
100
150
200
250
300
350
400
Latency (us) - Maximum 79 us (plotted on 01/31/13 at 01:16:44)



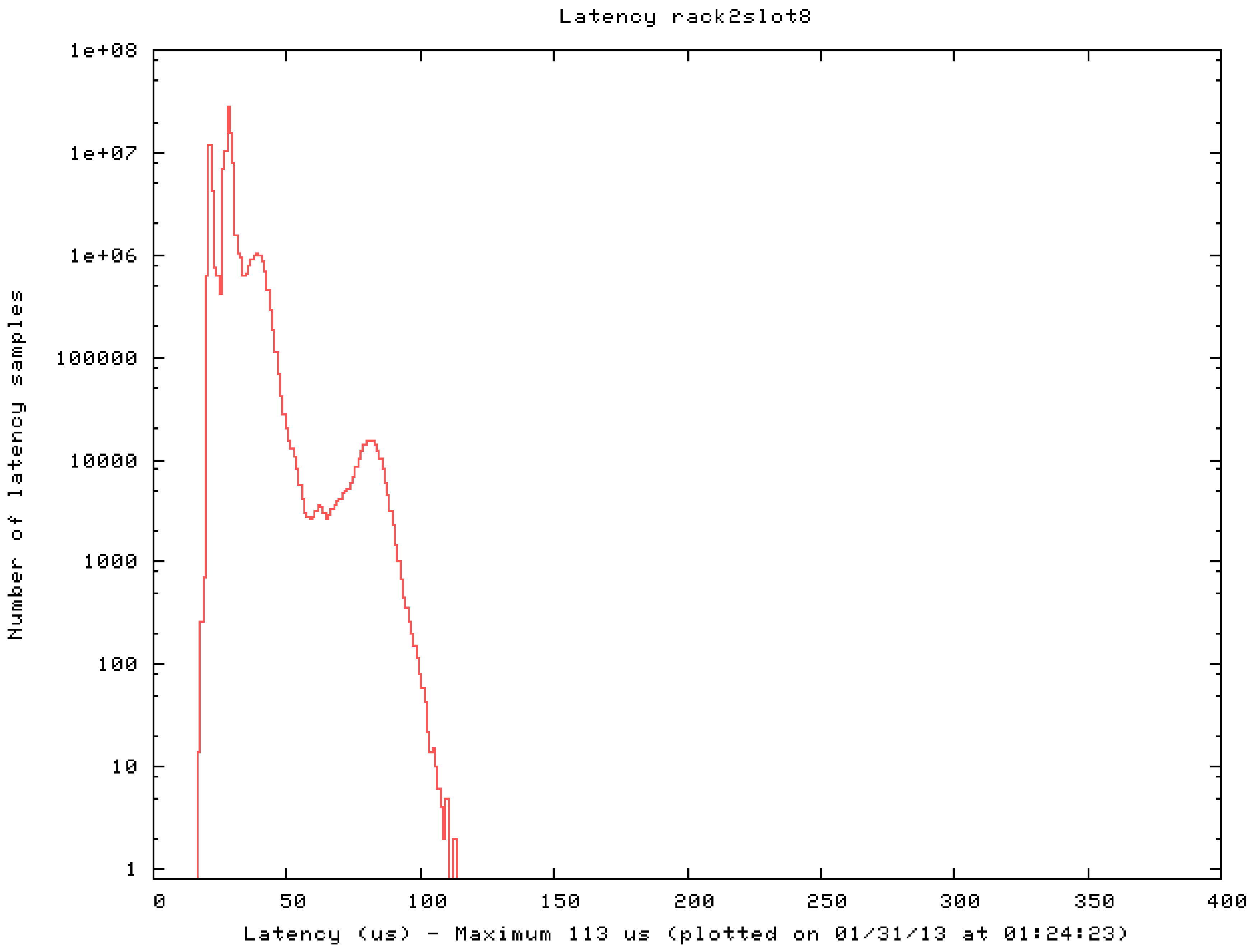
Latency rack1slot1
Number of latency samples
Latency (us) - Maximum 438 us (plotted on 01/29/13 at 12:44:18)
1e+08
1e+07
1e+06
100000
10000
1000
100
10
1
0
50
100
150
200
250
300
350
400
CPU0
CPU1
CPU2
CPU3
CPU4
CPU5
CPU6
CPU7
CPU8
CPU9
CPU10
CPU11
CPU12
CPU13
CPU14
CPU15
CPU16
CPU17
CPU18
CPU19
CPU20
CPU21
CPU22
CPU23
CPU24
CPU25
CPU26
CPU27
CPU28
CPU29
CPU30
CPU31


Latency rack2slot1
1e+07
1e+06
100000
10000
1000
100
10
1
Number of latency samples
0
50
100
150
200
250
300
350
400
Latency (us) - Maximum 389 us (plotted on 01/30/13 at 01:22:43)

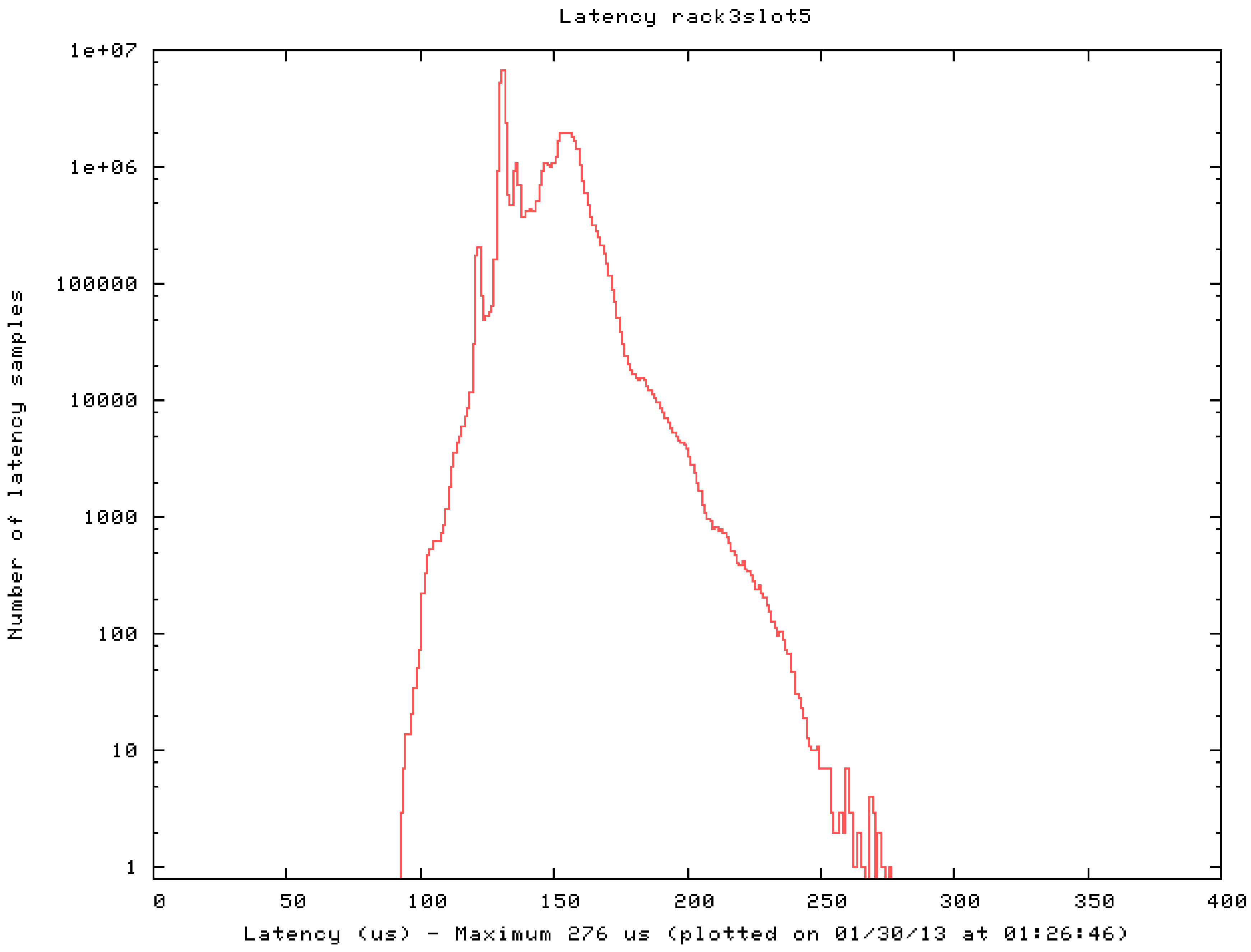

Latency rack2slot2
Number of latency samples
1e+06
100000
10000
1000
100
10
1
0
50
100
150
200
250
300
350
400
Latency (us) - Maximum 416 us (plotted on 01/30/13 at 01:22:55)

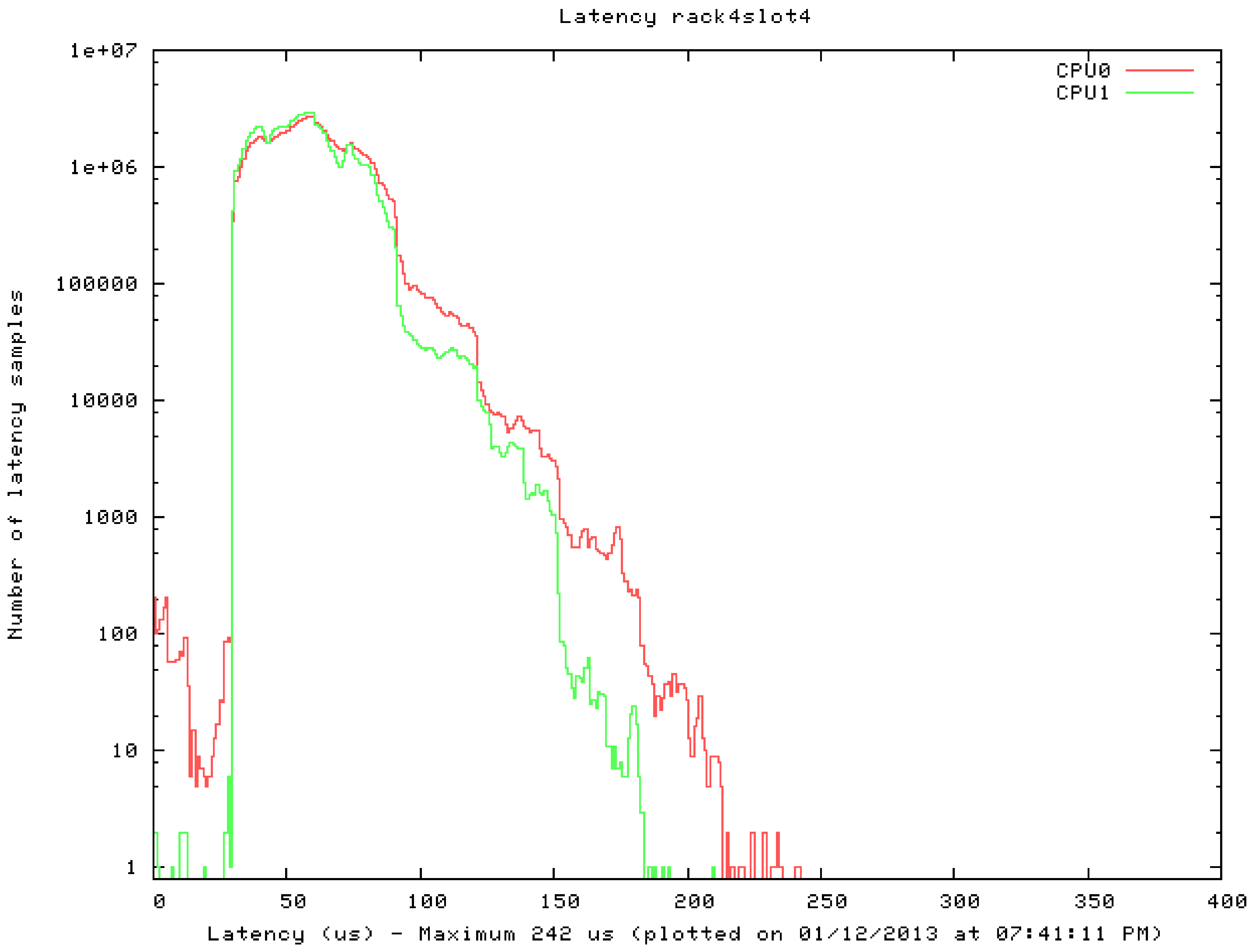


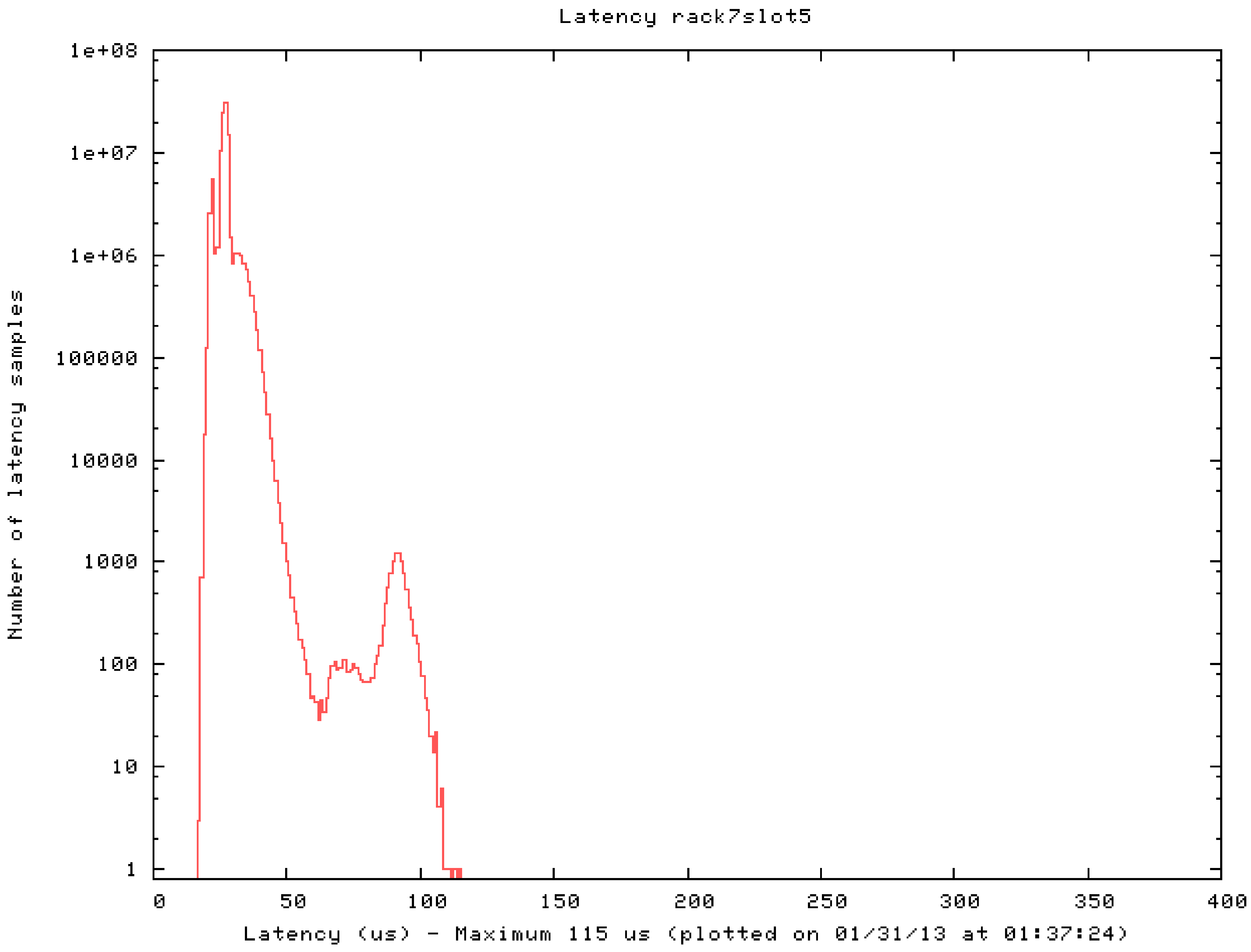
Latency rack2slot5
1e+08
1e+07
1e+06
100000
10000
1000
100
10
1
0
50
100
150
200
250
300
350
400
Number of latency samples
Latency (us) - Maximum 195 us (plotted on 01/30/13 at 01:23:48)



Latency rack2slot8
Number of latency samples
1e+08
1e+07
1e+06
100000
10000
1000
100
10
1
0
50
100
150
200
250
300
350
400
Latency (us) - Maximum 113 us (plotted on 01/31/13 at 01:24:23)

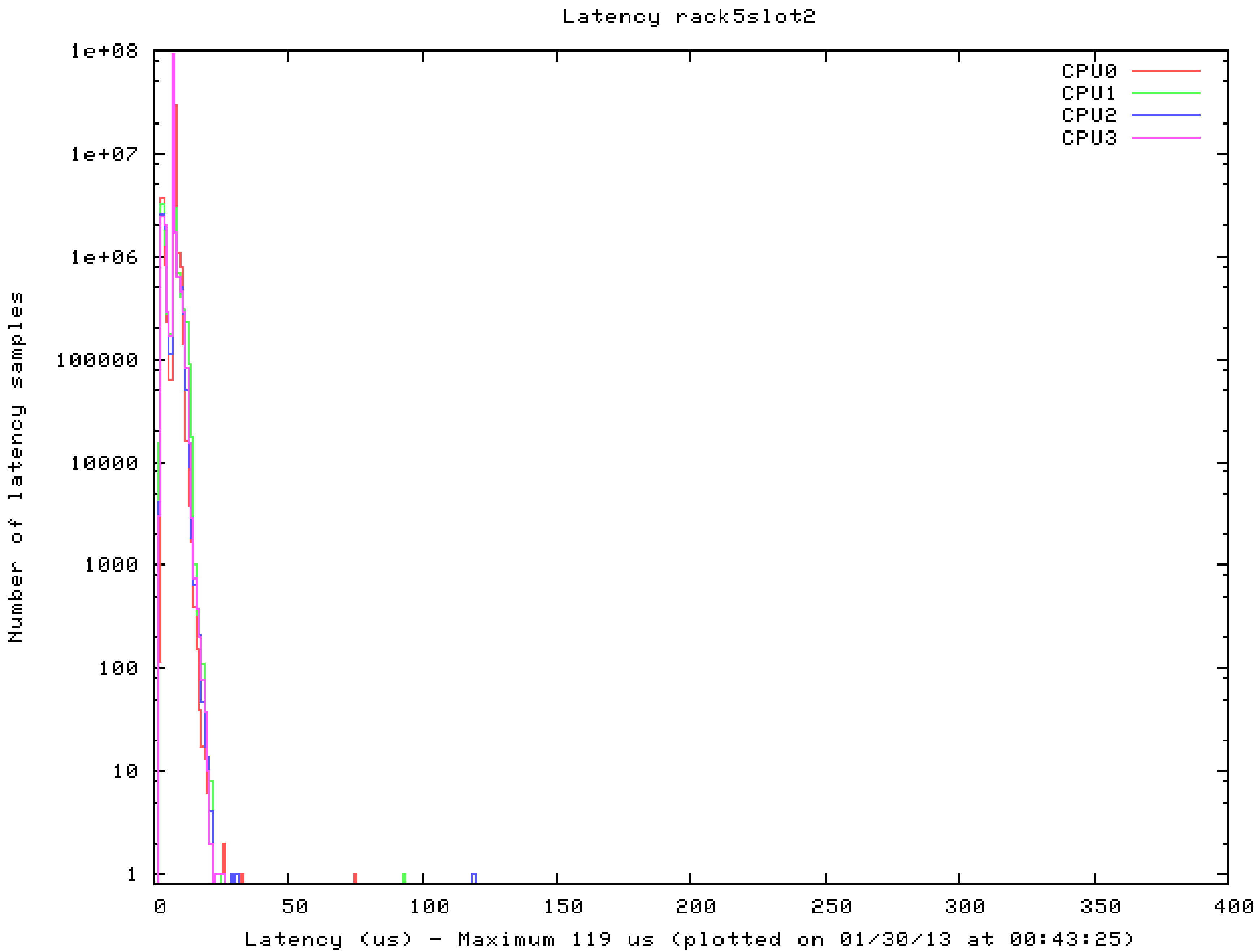

Latency rack3slot5
1e+07
1e+06
100000
10000
1000
100
10
1
Number of latency samples
0
50
100
150
200
250
300
350
400
Latency (us) - Maximum 276 us (plotted on 01/30/13 at 01:26:46)

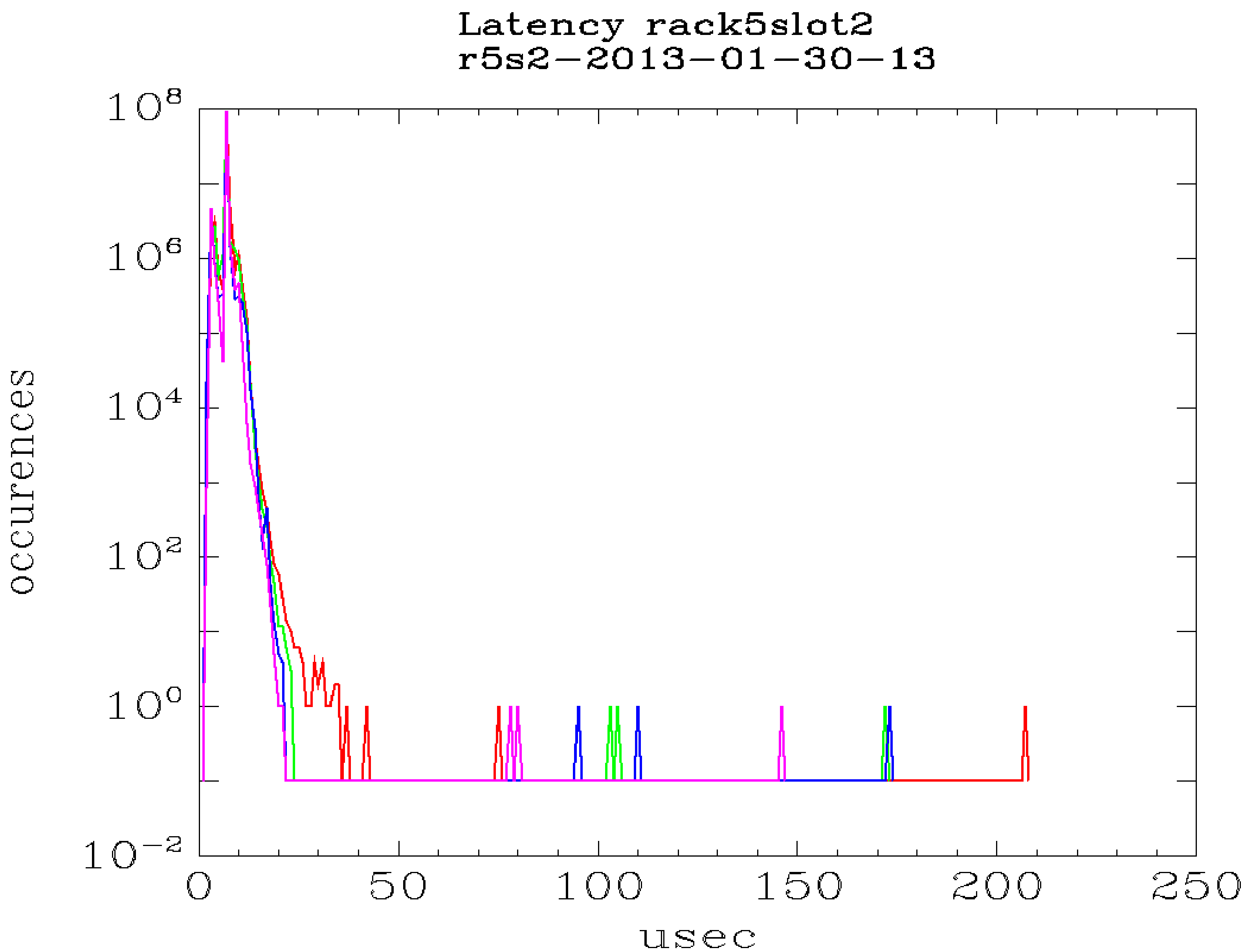

Latency rack4slot4
CPU0
CPU1
Number of latency samples
1e+07
1e+06
100000
10000
1000
100
10
1
0
50
100
150
200
250
300
350
400
Latency (us) - Maximum 242 us (plotted on 01/12/2013 at 07:41:11 PM)

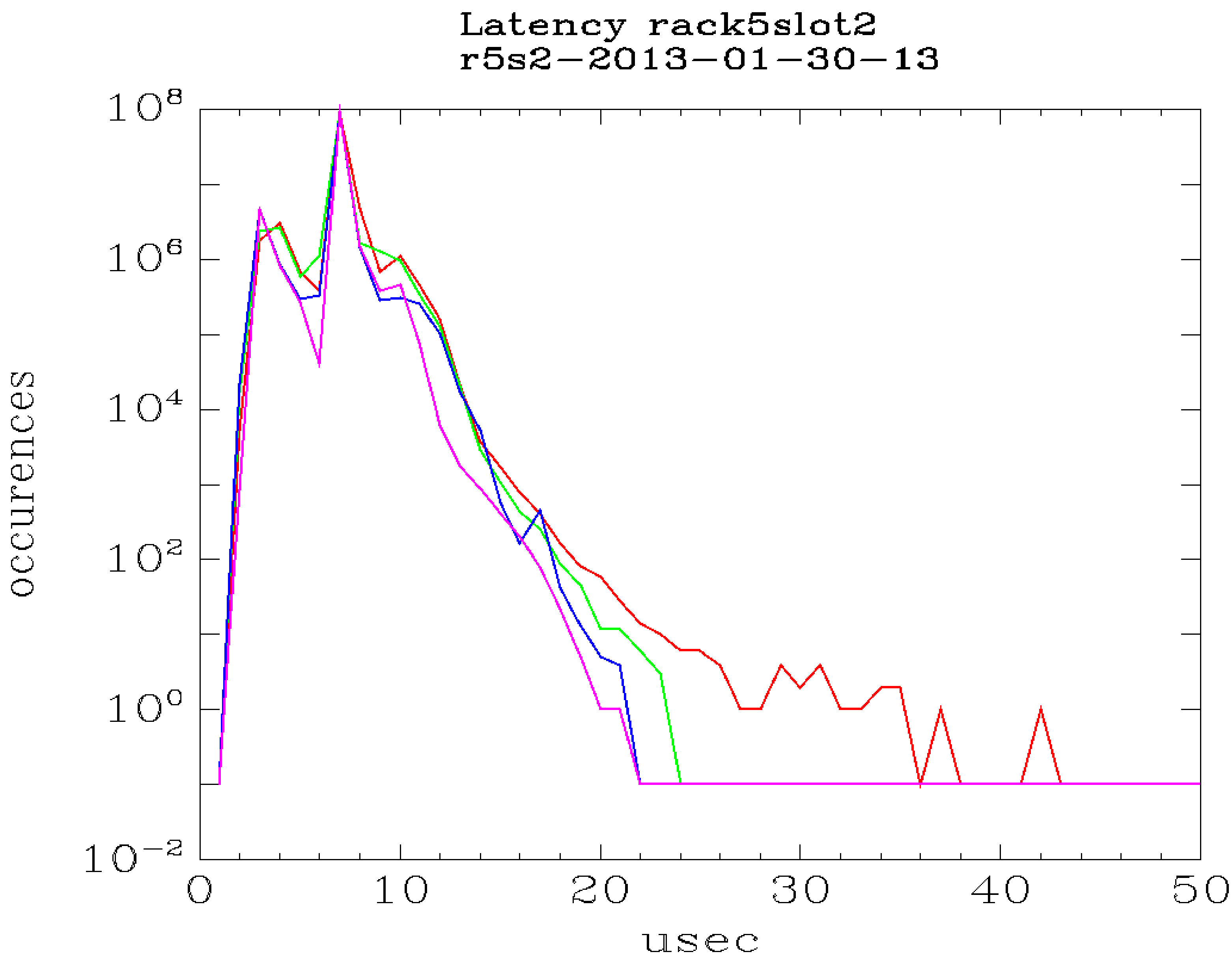


Latency rack7slot5
1e+08
1e+07
1e+06
100000
10000
1000
100
10
1
Number of latency samples
0
50
100
150
200
250
300
350
400
Latency (us) - Maximum 115 us (plotted on 01/31/13 at 01:37:24)

Even “boring” graphs may contain interesting details


Latency rack5slot2
CPU0
CPU1
CPU2
CPU3
1e+08
1e+07
1e+06
100000
10000
1000
100
10
1
Number of latency samples
0
50
100
150
200
250
300
350
400
Latency (us) - Maximum 119 us (plotted on 01/30/13 at 00:43:25)


Latency rack5slot2
r5s2-2013-01-30-13
occurences
usec
10^8
10^6
10^4
10^2
10^0
10^{-2}
0
50
100
150
200
250



Latency rack5slot2
r5s2-2013-01-30-13
occurences
usec
10^8
10^6
10^4
10^2
10^0
10^-2
0
10
20
30
40
50

Command Line Options

An unruly, out of control, set of control knobs

```
$ cyclictest  --help
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)
                           US is the max time to be be tracked in microseconds
-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       --threads         one thread per available processor
-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)
                           thread data structures allocated from local node
-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

```
-b USEC  --breaktrace=USEC send break trace command when latency > USEC
-B       --preemptirqs     both preempt and irqsoff tracing (used with -b)
-C       --context         context switch tracing (used with -b)
-E       --event           event tracing (used with -b)
-f       --ftrace          function trace (when -b is active)
-I       --irqsoff         Irqsoff tracing (used with -b)
-O TOPT  --traceopt=TOPT   trace option
-P       --preemptoff      Preempt off tracing (used with -b)
-R       --resolution      check clock resolution, calling clock_gettime() many
                           times.  list of clock_gettime() values will be
                           reported with -X
-T TRACE --tracer=TRACER   set tracing function
    configured tracers: blk function_graph wakeup_rt wakeup function nop
-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
```

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 -l100 -a0
$ cyclictest -t -l100 -a 0
$ cyclictest -t -l100 -a7 -a0
```

affinity will be 7, with no error message

```
$ cyclictest -t -l100 -a7 -a 0
```

```
-a  cpu affinity
```

Options Trivia

Options parsing is not robust - example 2

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

$ cyclictest -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

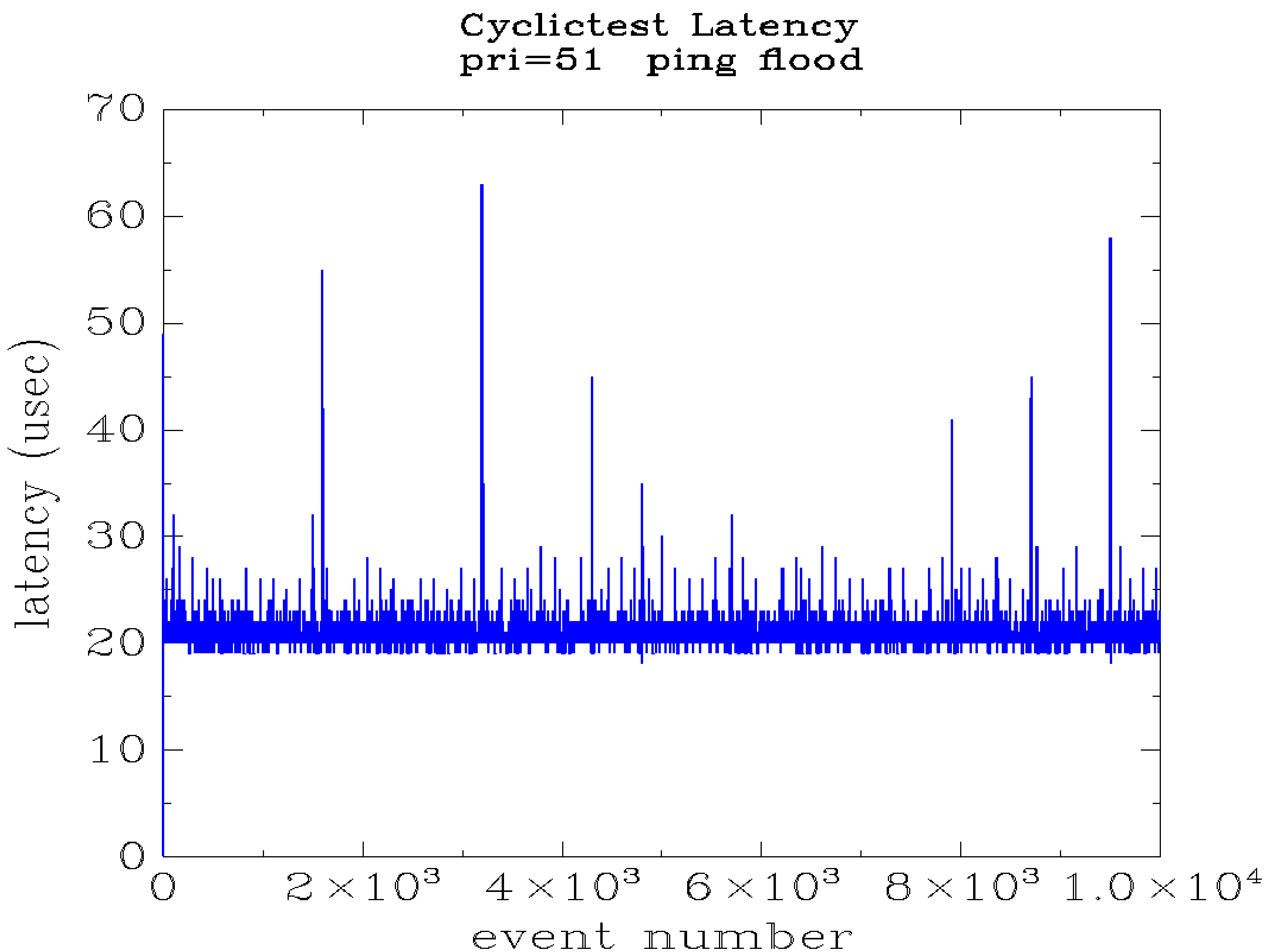
Best Practice:

- 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
```



Cyclictest Latency
pri=51 ping flood
latency (usec)
event number
0
10
20
30
40
50
60
70
2×10³
4×10³
6×10³
8×10³
1.0×10⁴

Hitting the RT sched throttle

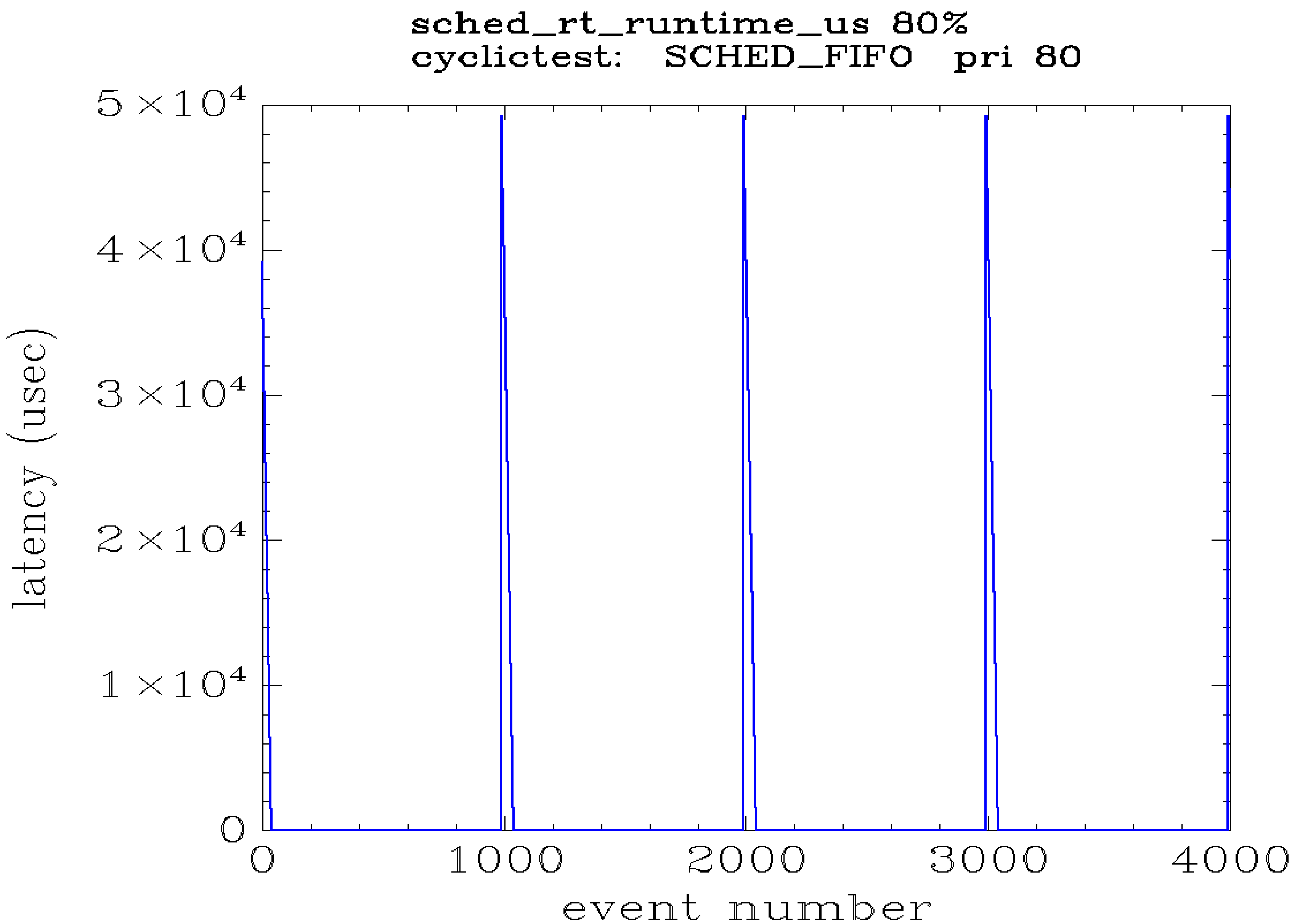
/proc/sys/kernel/sched_rt_runtime_us

/proc/sys/kernel/sched_rt_period_us

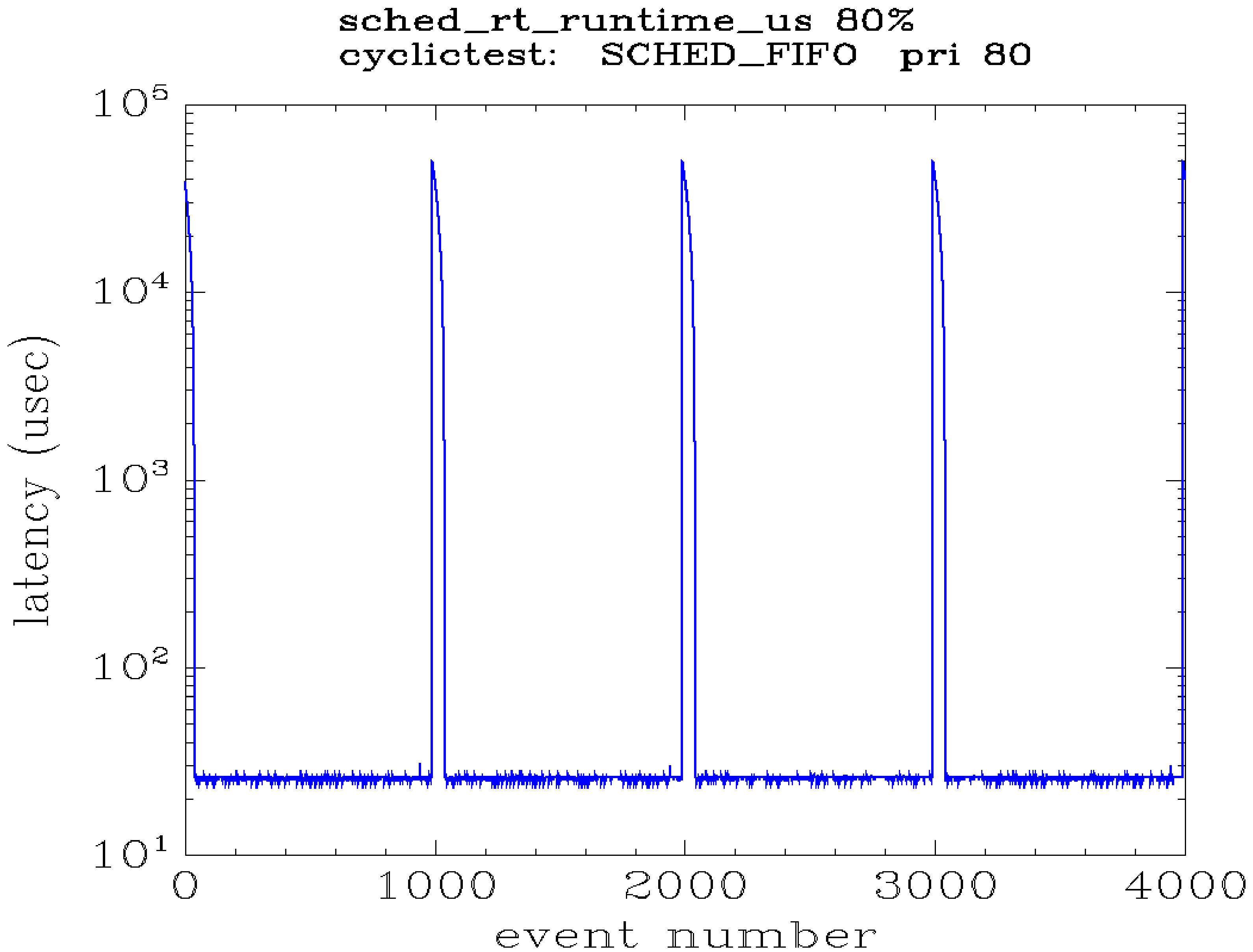
cyclictest: SCHED_FIFO priority=80

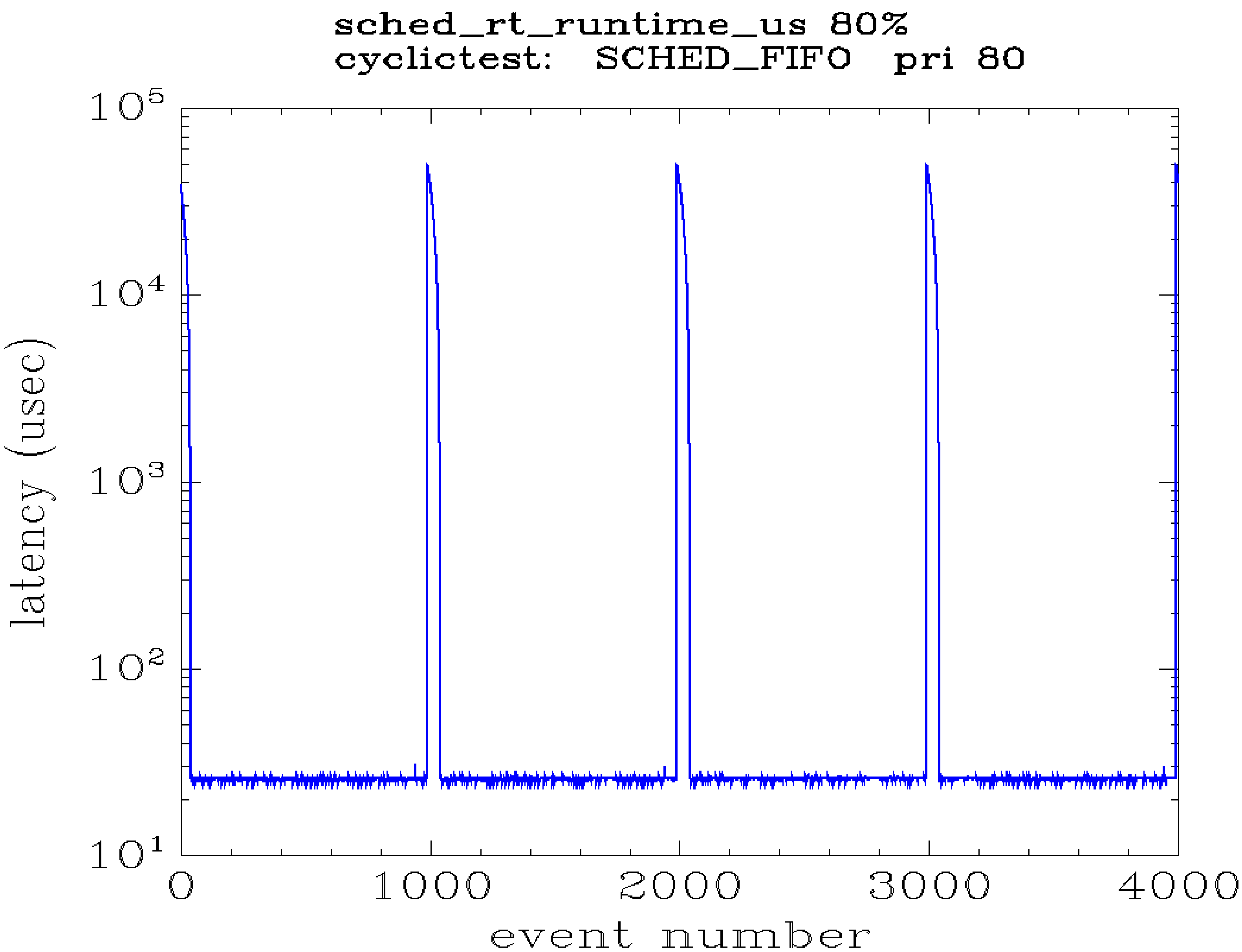
background load:

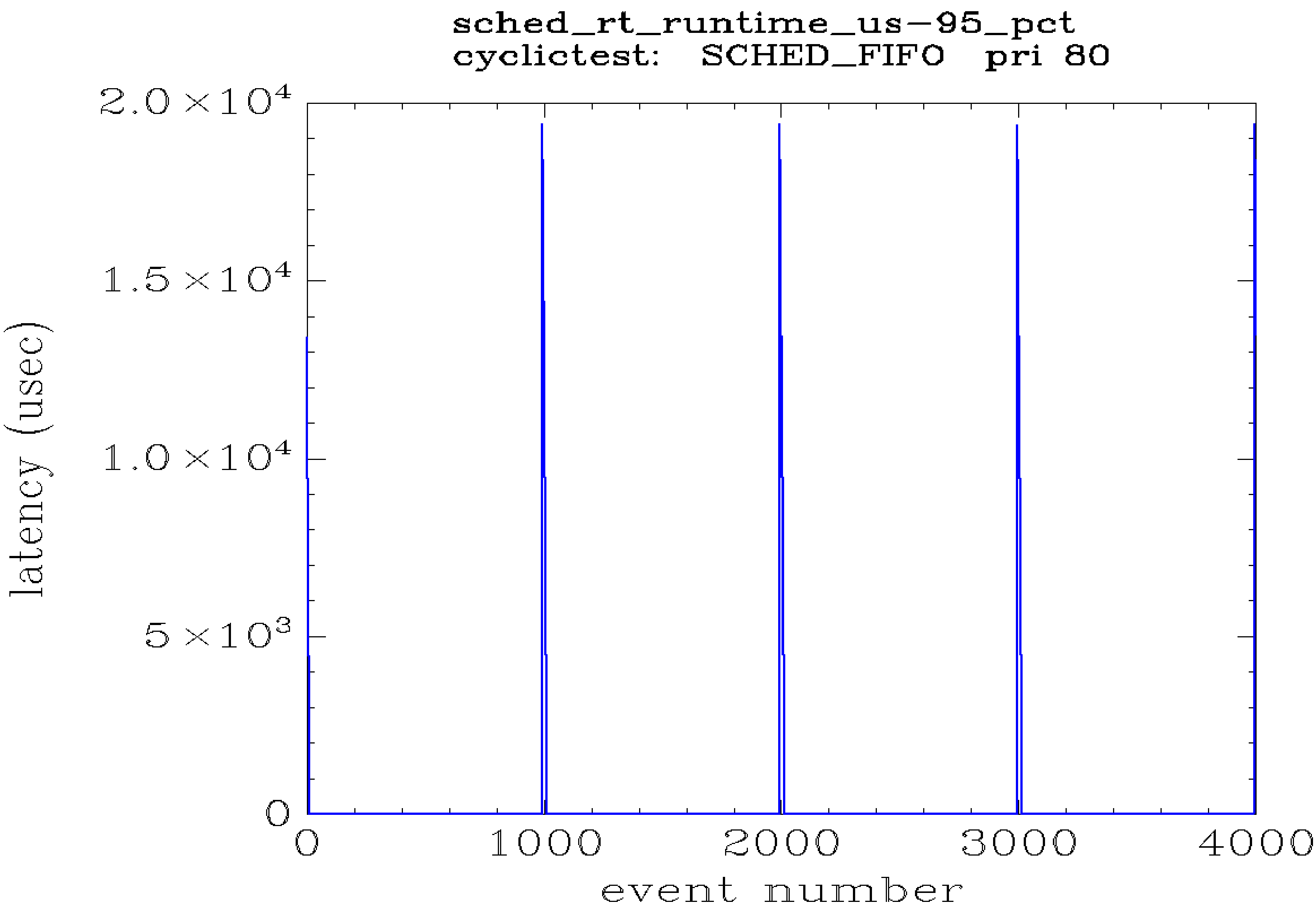
- continuous
- SCHED_FIFO priority=40

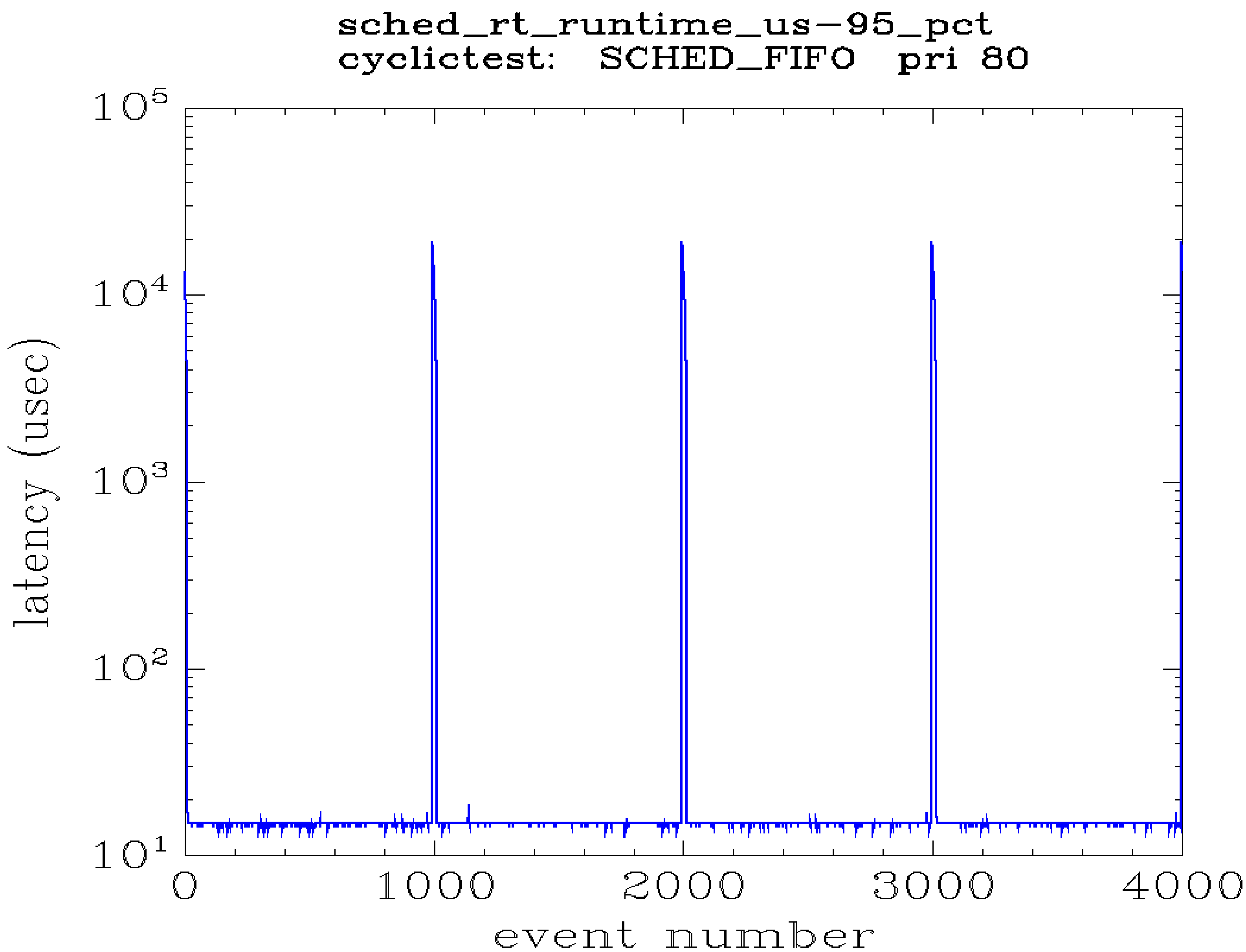


sched_rt_runtime_us 80%
cyclictest: SCHED_FIFO pri 80
latency (usec)
5×10^4
4×10^4
3×10^4
2×10^4
1×10^4
0
0
1000
2000
3000
4000
event number


sched_rt_runtime_us 80%
cyclictest: SCHED_FIFO pri 80
latency (usec)
10^5
10^4
10^3
10^2
10^1
0
1000
2000
3000
4000
event number


sched_rt_runtime_us 80%
cyclictest: SCHED_FIFO pri 80
10^5
10^4
10^3
10^2
10^1
latency (usec)
0
1000
2000
3000
4000
event number


sched_rt_runtime_us-95_pct
cyclictest: SCHED_FIFO pri 80
2.0×10⁴
1.5×10⁴
1.0×10⁴
5×10³
0
latency (usec)
0
1000
2000
3000
4000
event number


sched_rt_runtime_us-95_pct
cyclictest: SCHED_FIFO pri 80
latency (usec)
10^5
10^4
10^3
10^2
10^1
0
1000
2000
3000
4000
event number

Hitting the RT sched throttle

/proc/sys/kernel/sched_rt_runtime_us

/proc/sys/kernel/sched_rt_period_us

cyclictest: SCHED_NORMAL

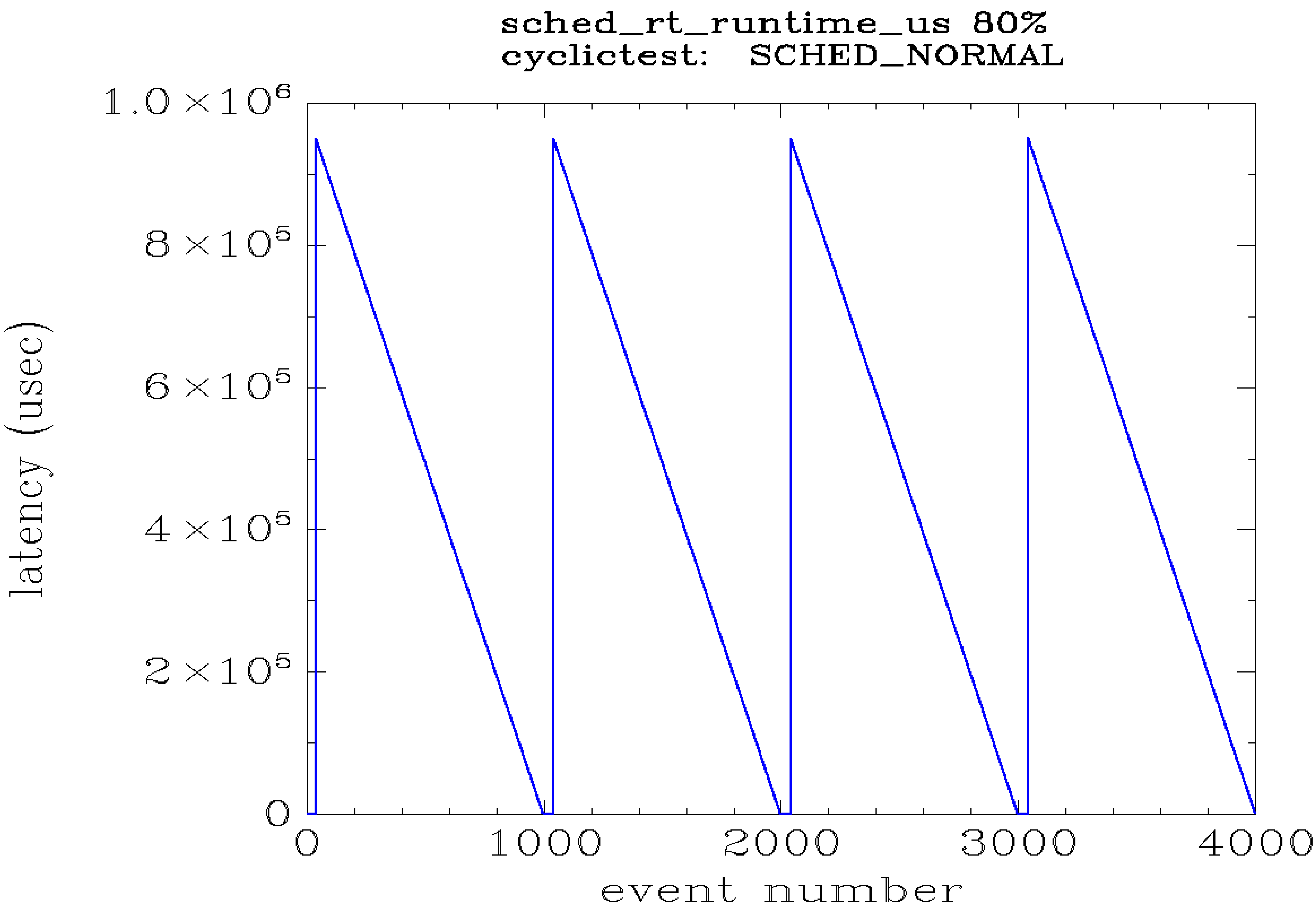
background load:

- continuous
- SCHED_FIFO priority=40

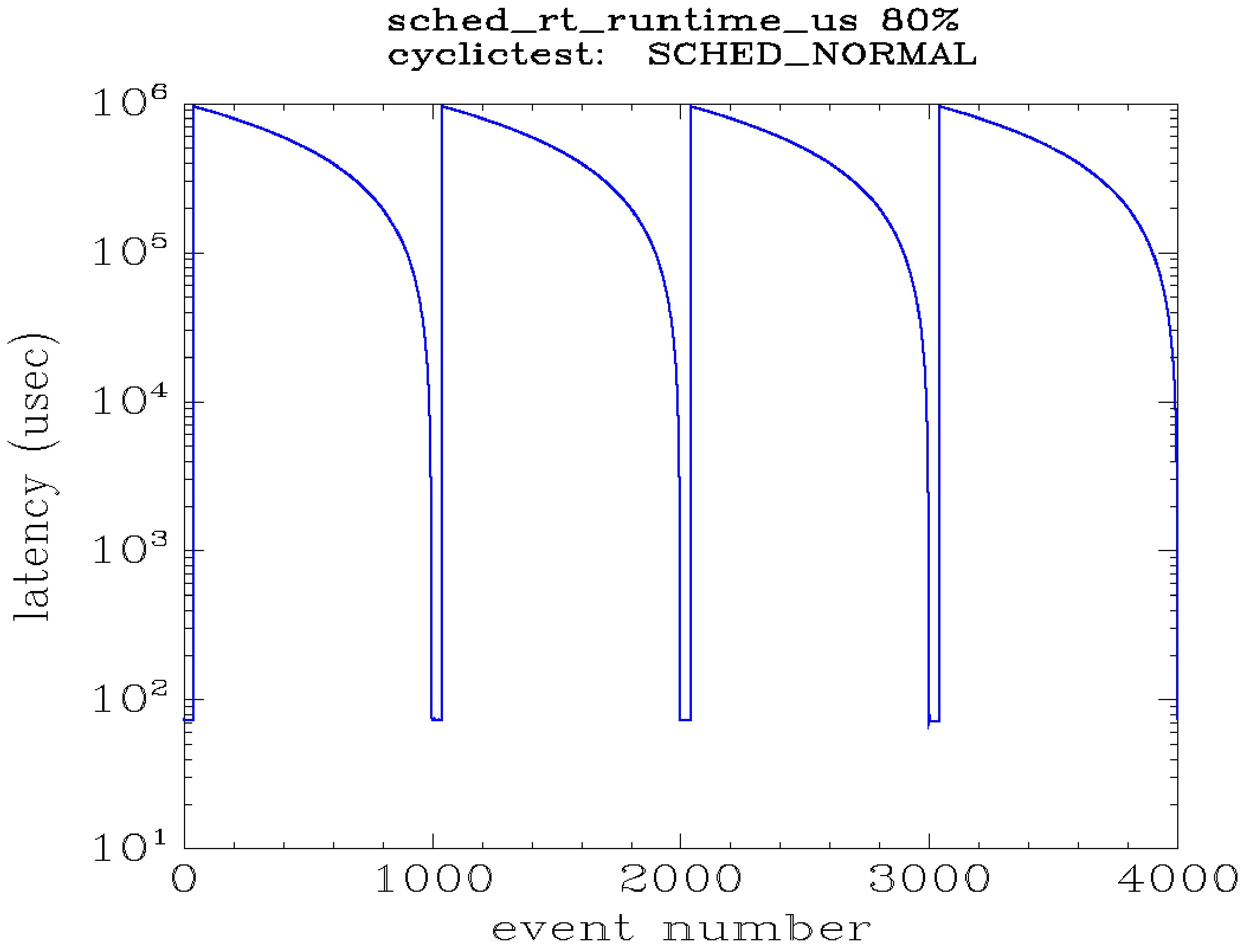
Hitting the RT sched throttle

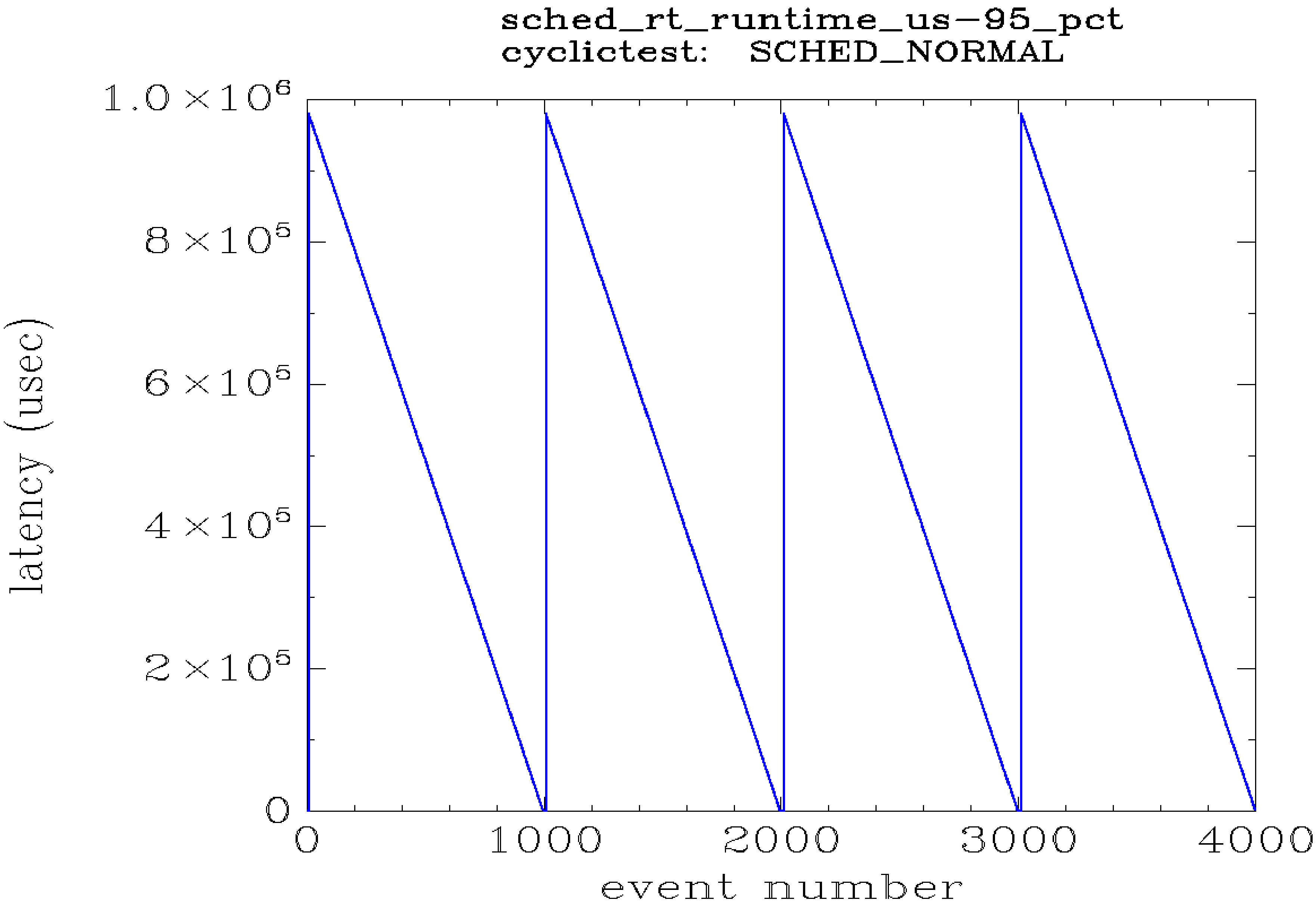
Why is this measurement interesting???

Gives a picture of how much cpu is NOT used by the real time tasks

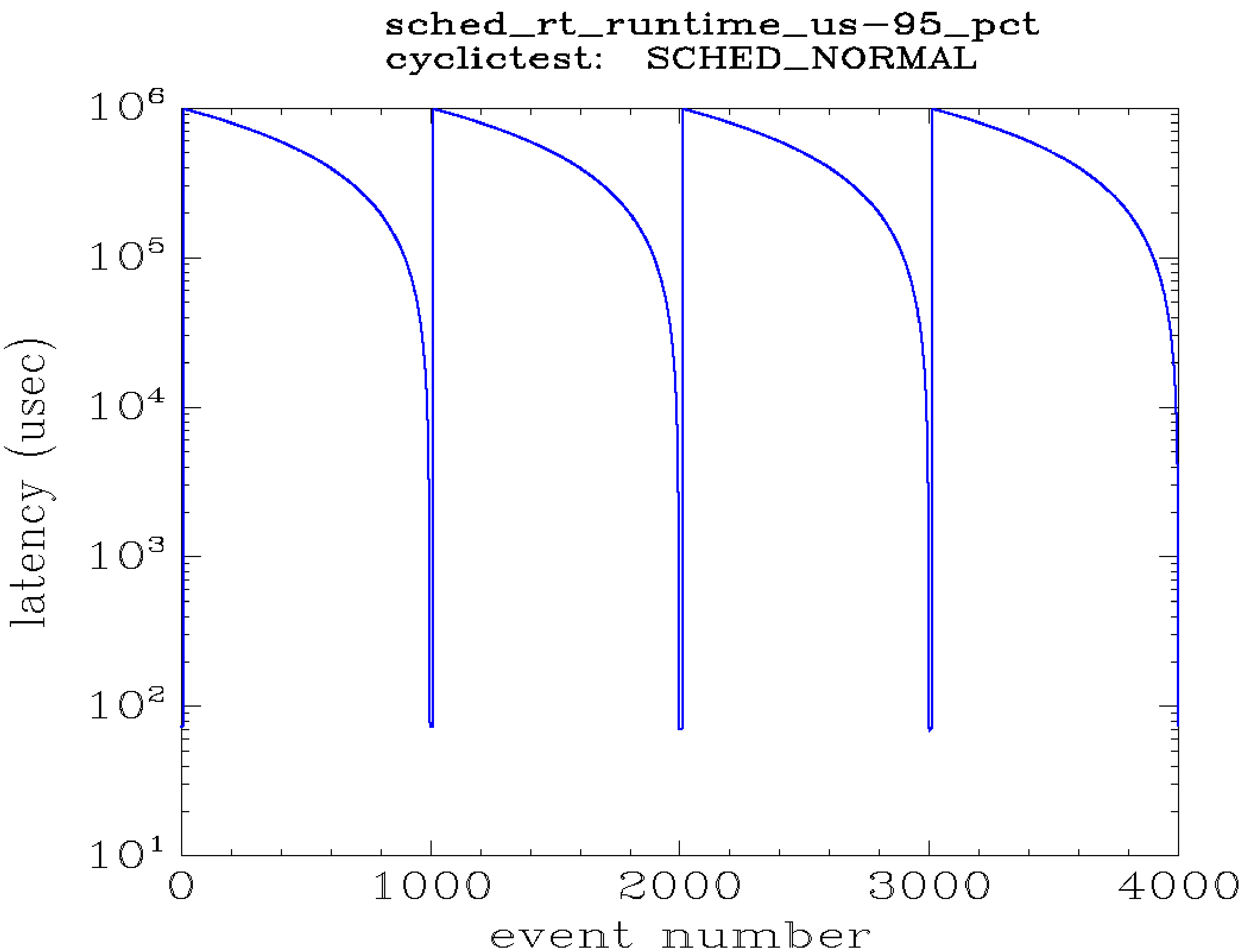


sched_rt_runtime_us 80%
cyclictest: SCHED_NORMAL
1.0 ×10^6
8 ×10^5
6 ×10^5
4 ×10^5
2 ×10^5
0
latency (usec)
0
1000
2000
3000
4000
event number


sched_rt_runtime_us 80%
cyclictest: SCHED_NORMAL
latency (usec)
event number
0
1000
2000
3000
4000
10^1
10^2
10^3
10^4
10^5
10^6



sched_rt_runtime_us-95_pct
cyclictest: SCHED_NORMAL
1.0×10^6
8×10^5
6×10^5
4×10^5
2×10^5
0
latency (usec)
0
1000
2000
3000
4000
event number


sched_rt_runtime_us=95_pct
cyclictest: SCHED_NORMAL
latency (usec)
10^6
10^5
10^4
10^3
10^2
10^1
0
1000
2000
3000
4000
event number

Unusual Uses of Cyclictest

Rough measurement of response time of a real time application, without instrumenting task.

cyclictest latency =~

task latency + task work duration

This is not an accurate measurement, but it does provide a rough picture.

Response Time of a Task

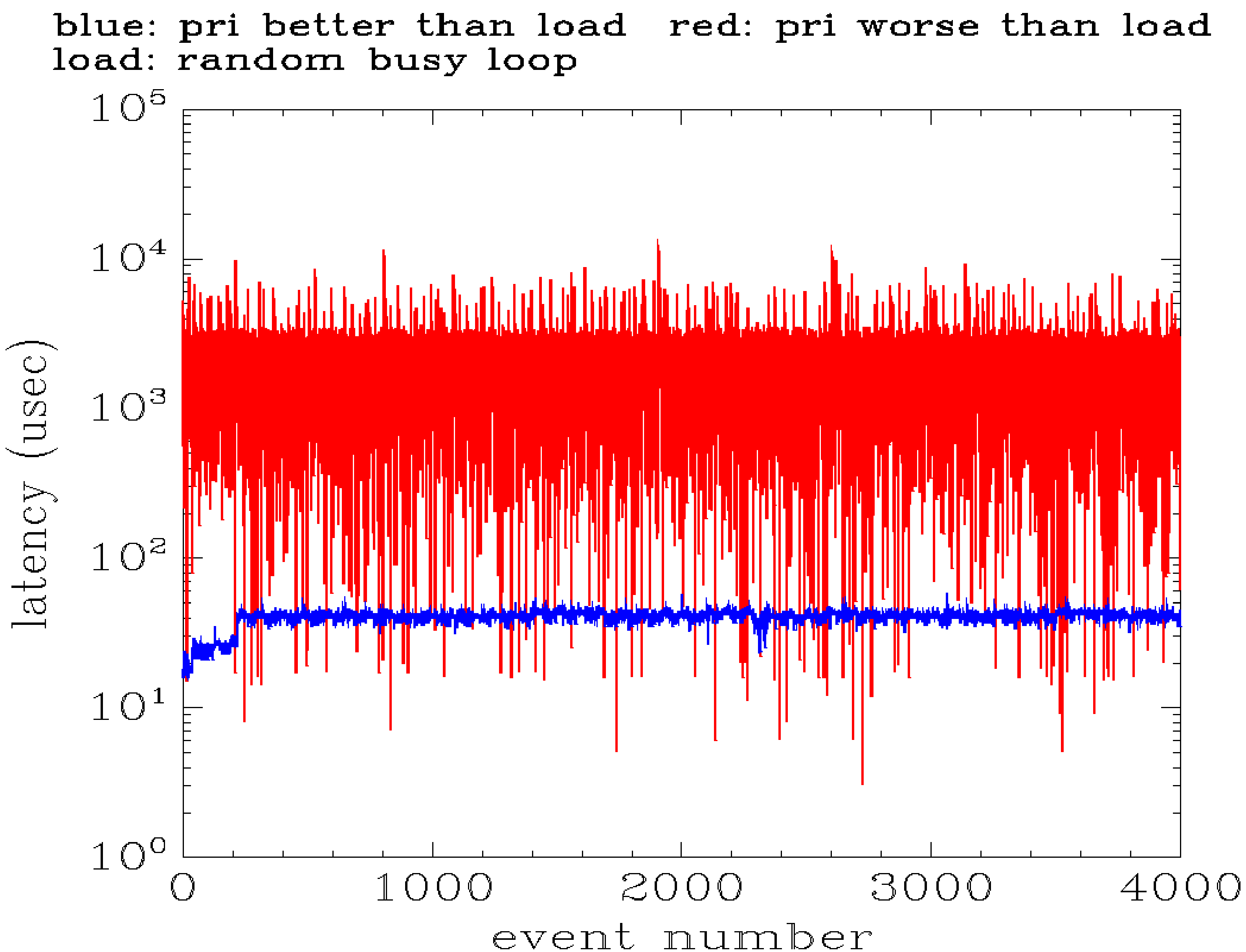
Cyclictest:

(1) SCHED_FIFO priority=80
baseline latency

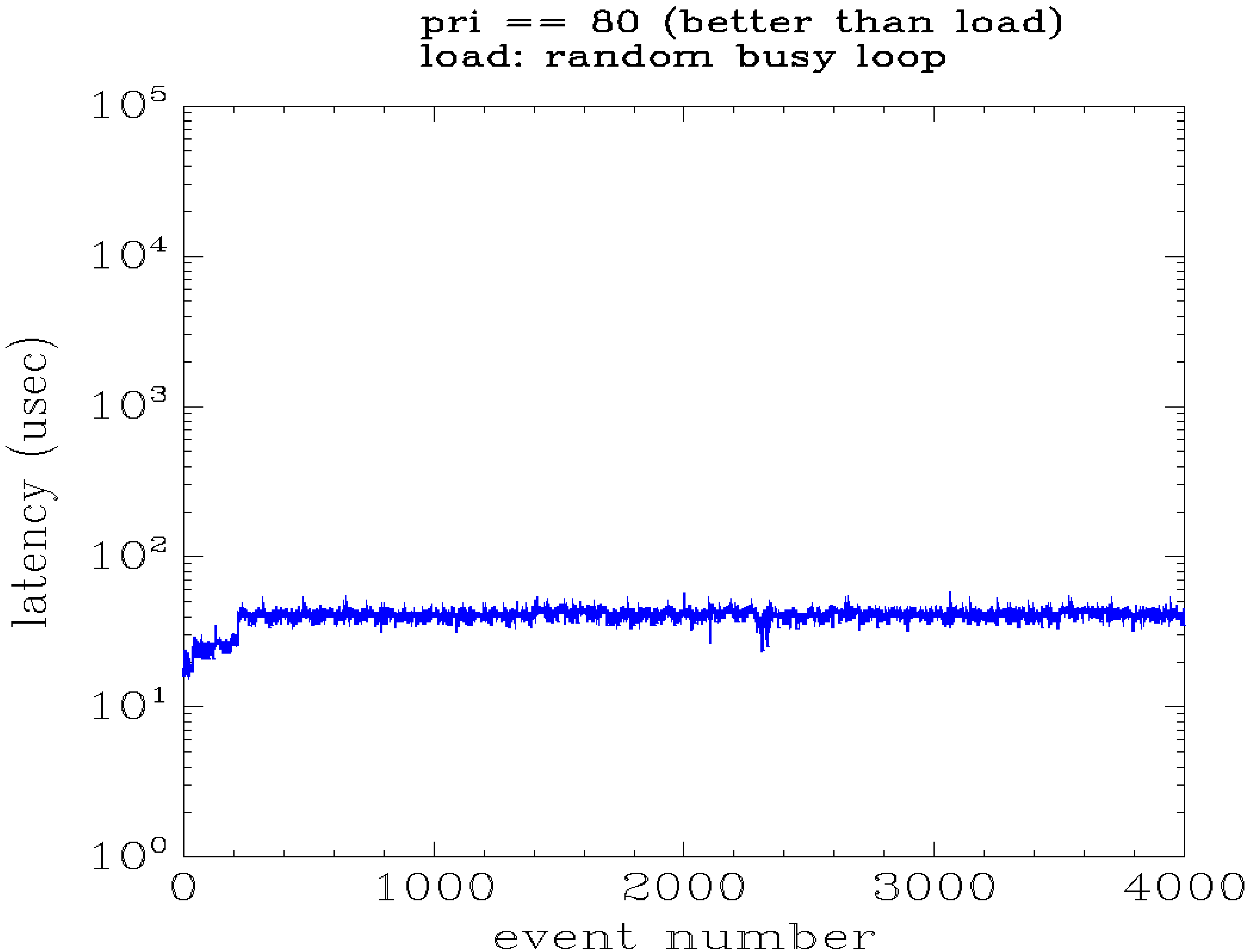
(2) SCHED_FIFO priority=30
approximation of task response time

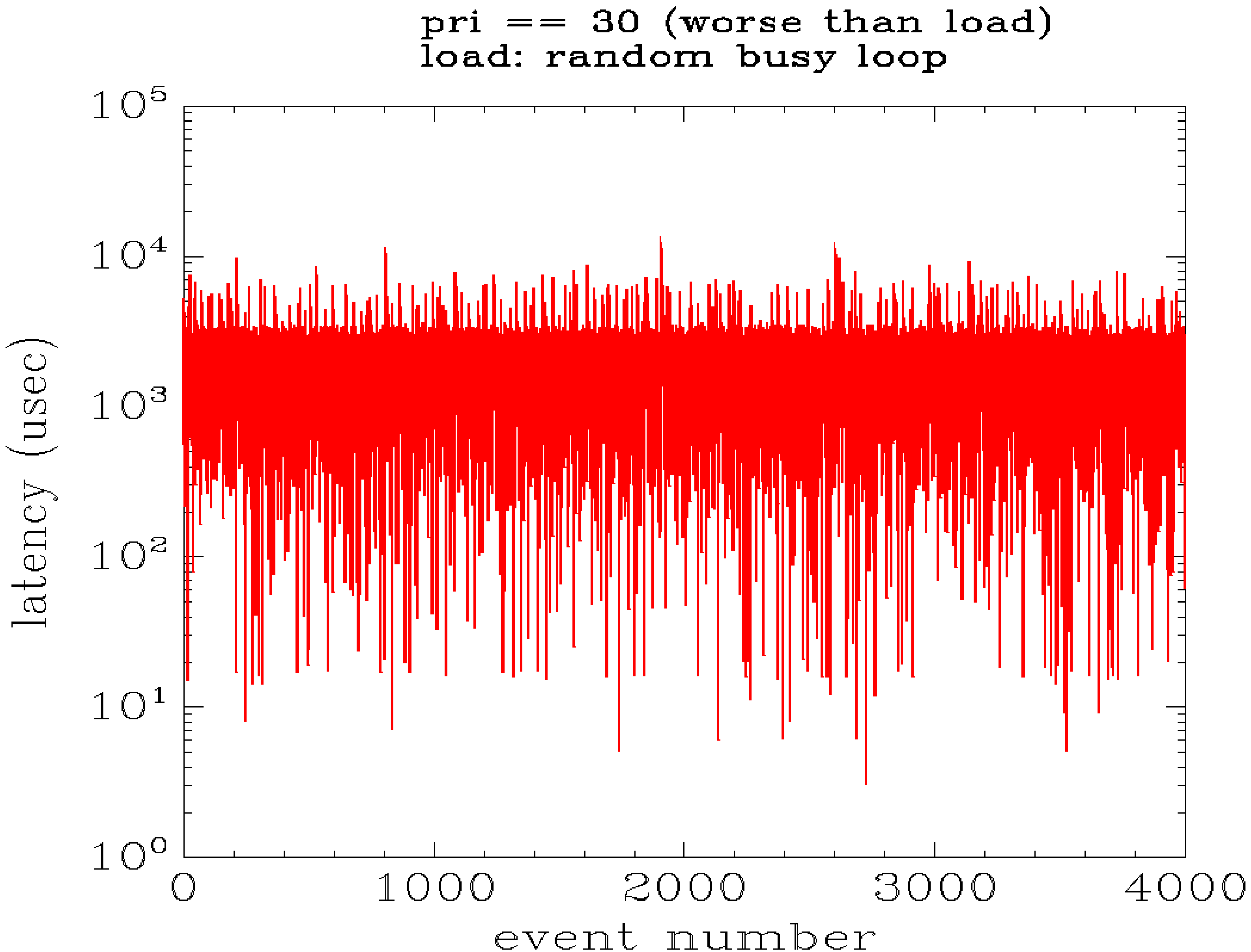
Real time application:

- busy loop (random number of iterations), followed by a short sleep
- SCHED_FIFO priority=40



blue: pri better than load red: pri worse than load
load: random busy loop
latency (usec)
event number
10^0
10^1
10^2
10^3
10^4
10^5
0
1000
2000
3000
4000


pri == 80 (better than load)
load: random busy loop
latency (usec)
event number
10^0
10^1
10^2
10^3
10^4
10^5
0
1000
2000
3000
4000



pri == 30 (worse than load)
load: random busy loop
latency (usec)
10^0
10^1
10^2
10^3
10^4
10^5
0
1000
2000
3000
4000
event number

Response Time of a Task

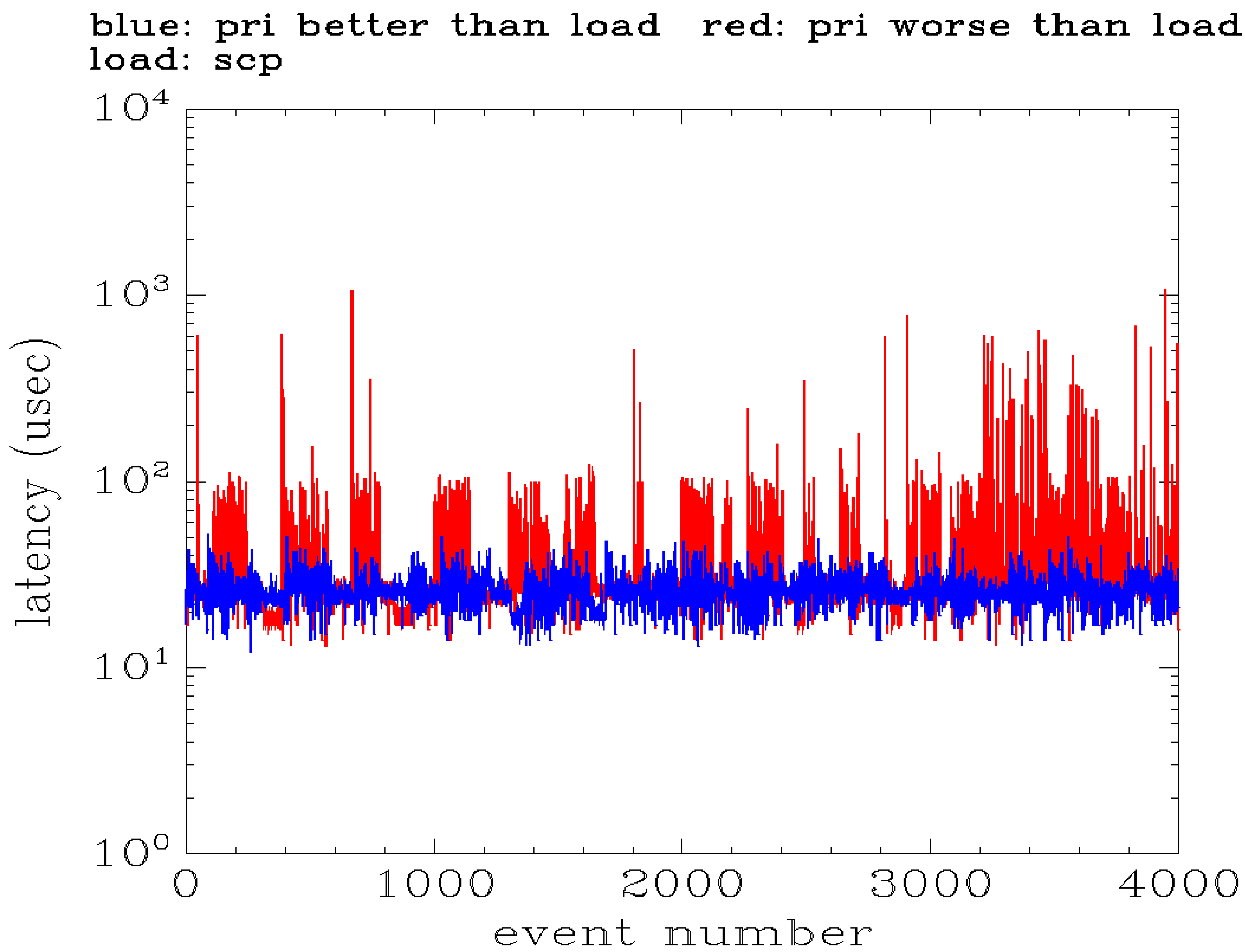
Cyclictest:

(1) SCHED_FIFO priority=80
baseline latency

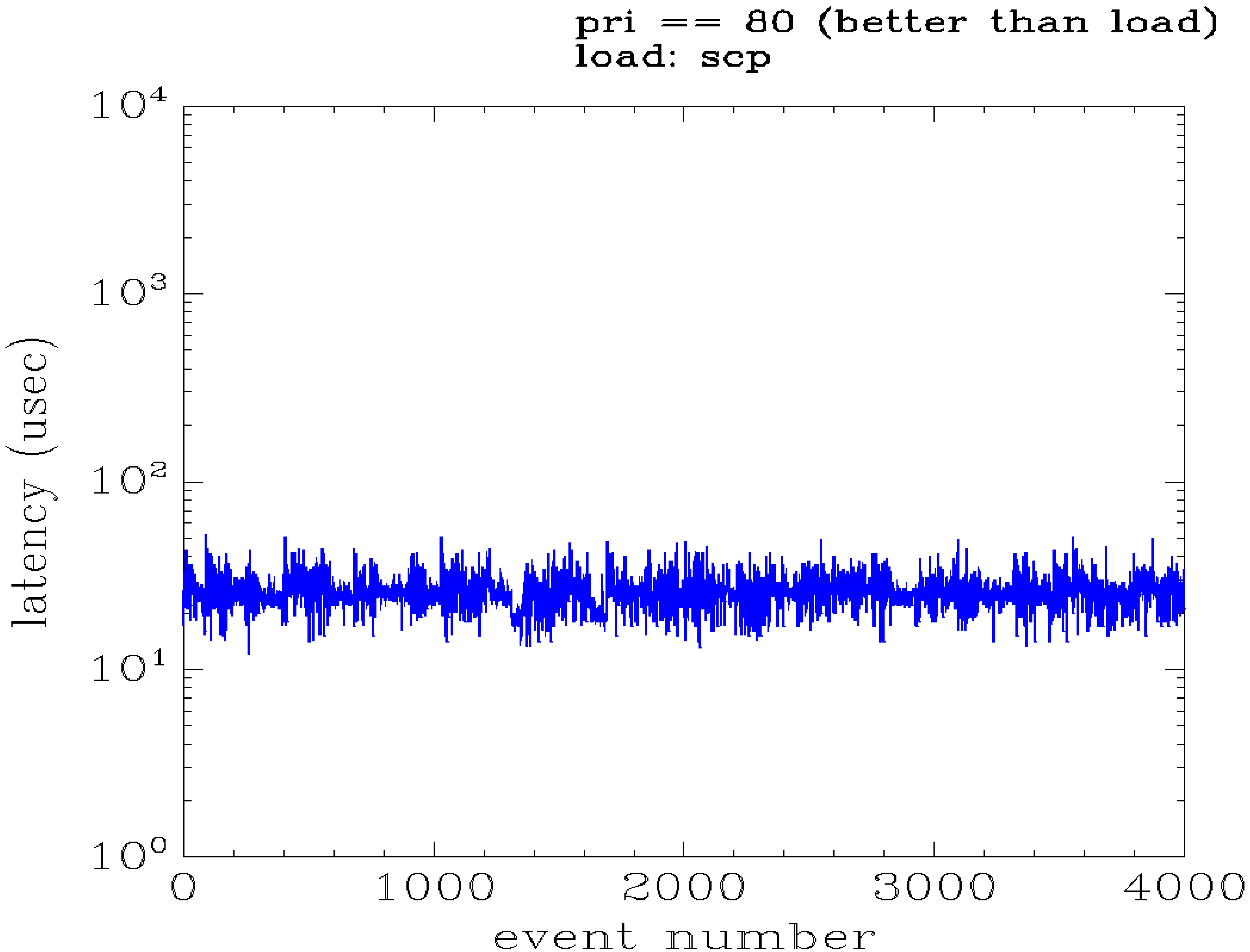
(2) SCHED_FIFO priority=30
approximation of task response time

Real time application:

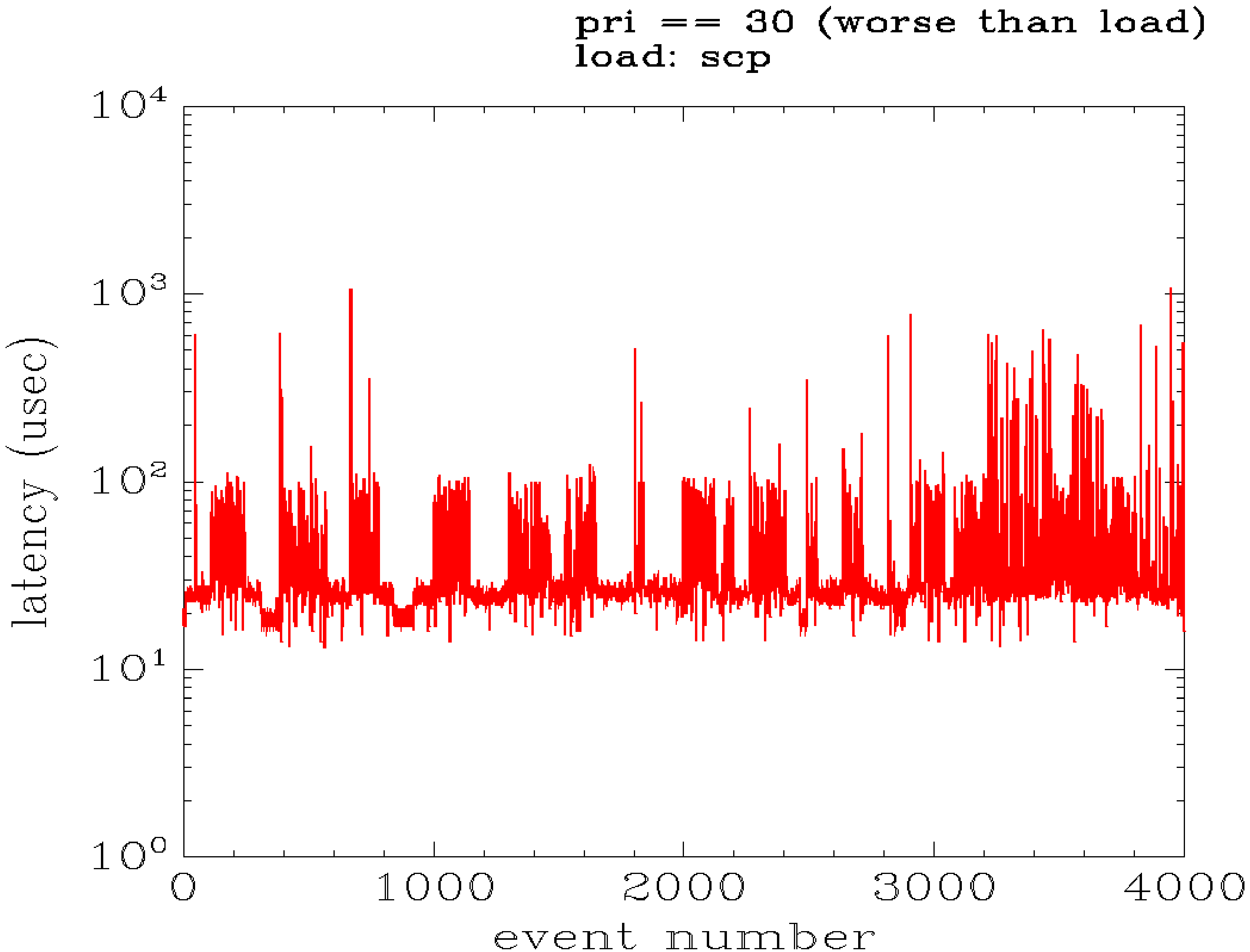
- recursive scp of a panda root file system
- SCHED_FIFO priority=40
- no guarantee of sleep between scp "transactions" - response time may include multiple transactions



blue: pri better than load red: pri worse than load
load: scp
latency (usec)
event number
10^4
10^3
10^2
10^1
10^0
0
1000
2000
3000
4000



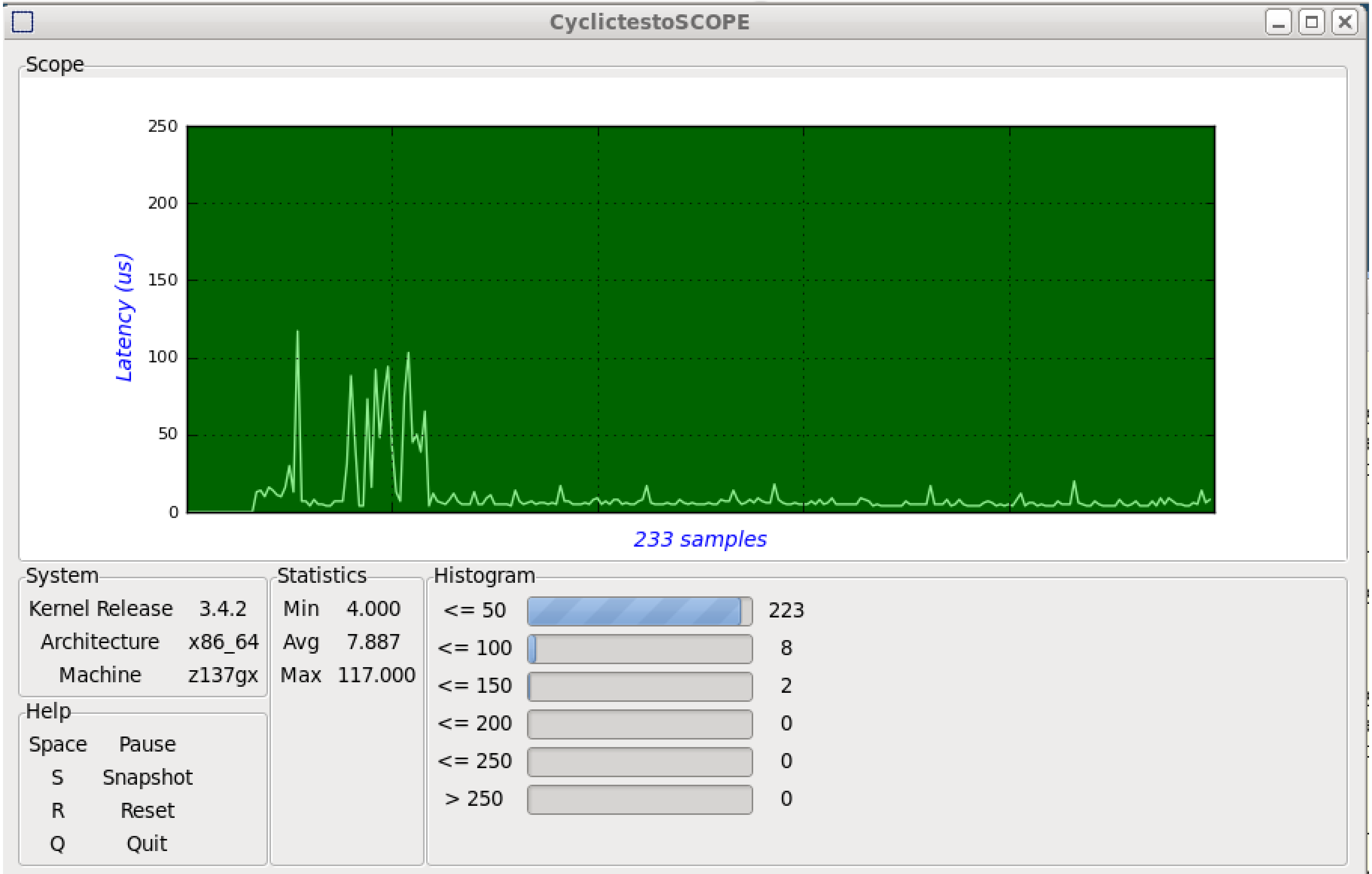
pri == 80 (better than load)
load: scp
latency (usec)
event number
10^0
10^1
10^2
10^3
10^4
0
1000
2000
3000
4000


pri == 30 (worse than load)
load: scp
latency (usec)
event number
10^4
10^3
10^2
10^1
10^0
0
1000
2000
3000
4000

Demo - oscilloscope

```
cyclictest_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@sonymobile.com