Adventures In Real-Time Performance Tuning

The real-time for Linux patchset does not guarantee adequate real-time behavior for all target platforms. When using real-time Linux on a new platform you should expect to have to tune the kernel and drivers to provide performance that matches your specific requirements.

This paper provides an example of the trials and tribulations of the tuning journey for a MIPS target board. A brief back of the envelope real-time performance characterization of the board will also be presented.

April 15, 2008

Adventures In Real-Time Performance Tuning

What I'm trying to tune

Some examples of using the available tuning tools

If I talk fast enough, some performance data for a MIPS TX4937 processor

What is Real Time?

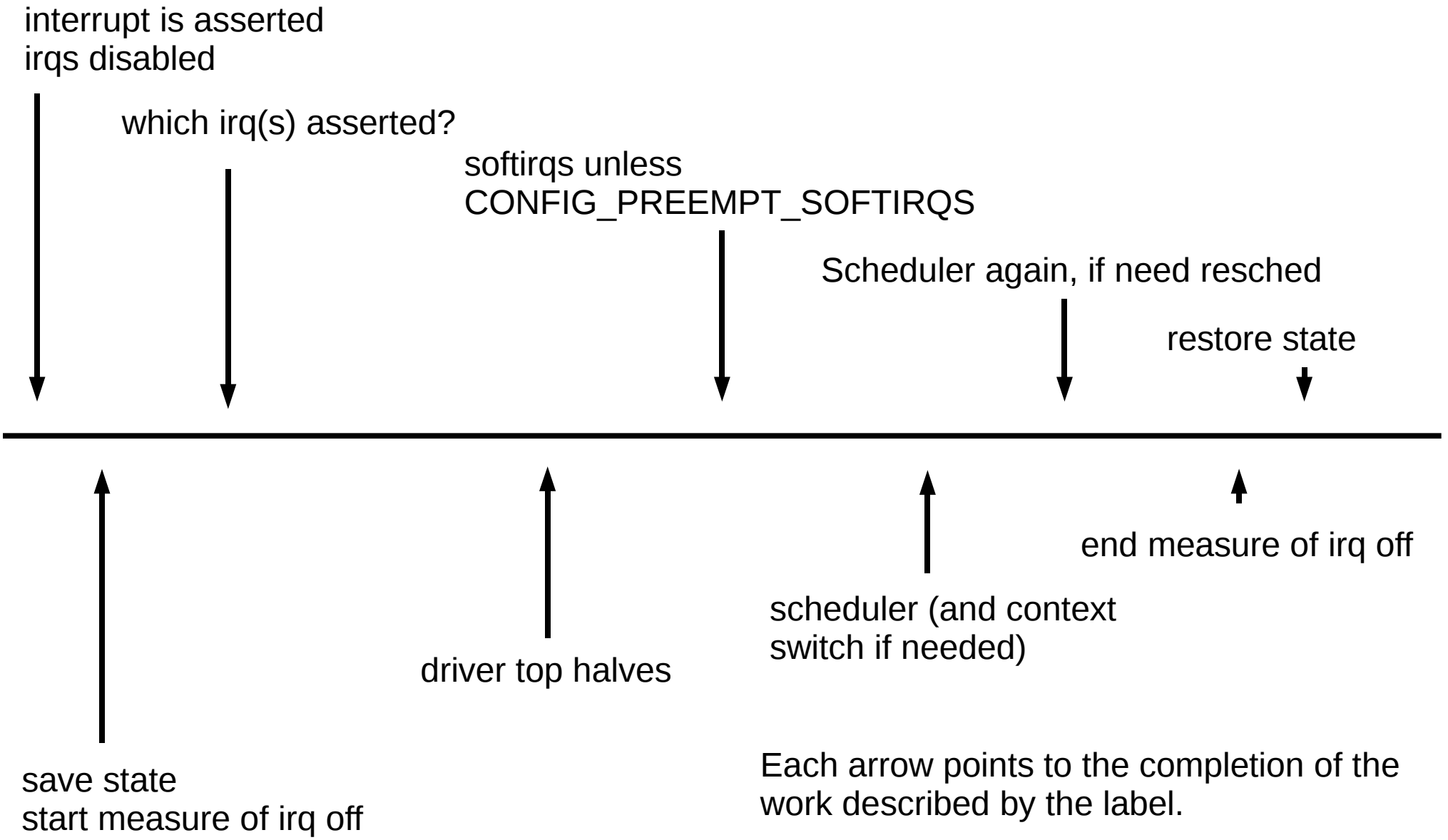
It is determinism (being able to respond to a stimulus before a deadline) within a given system load envelope.

It is NOT fast response time.

The specific real time application deadlines determine how short the maximum response time must be to deliver real time behaviour.

Some examples of deadlines are one second, one millisecond, or five microseconds.

irq off latency



Another key impact on RT latency is preempt off latency

I will ignore preempt off latency in this talk, but essentially the same methods used to tune irq off latency are used to tune preempt off latency

Actual RT latency is when the RT “application” is actually executing code.

The components that add up to RT latency are important to the tuning process, but keep in mind the goal of tuning actual RT latency.

Examples of Tuning Knobs

Which hardware is enabled and used
Which kernel functionality is enabled and used
Which drivers are used
Kernel config options
Softirq handling in thread context
Driver top half algorithms
Driver top half polling (vs irq)
Driver bottom half in thread context
Driver bottom half thread priorities
Real-time thread priorities
Non-RT application load

Examples of Tuning Knobs

Kernel algorithms and code
Locks
Timers
CPU affinity and partitioning
- drivers
- kernel threads
- user processes and threads

Lock code and data in memory
Lock tlb entries
Lock code and data in cache or fast memory

A Roadmap of my Journey

1) add some RT pieces for MIPS and the tx4937 processor

2) add MIPS support to RT instrumentation

3) tuning

4) implement “lite” irq disabled instrumentation

5) tuning

Caveats

Tools, instrumentation, techniques, etc are very dependent upon the version of the kernel and rt-preempt patcheset.

Kernel data structures, algorithms, performance hot spots change.

An example of instrumentation change:

RT tracing mechanism is in the process of being submitted to mainline (but is not in 2.6.25-rc8), which prompted partial re-write.

To follow this, see LKML:

From: Ingo Molnar
Date: Fri Feb 08 2008
Subject: [git pull] latency tracer
“Linus, please pull the latency tracer tree”

Date: Fri Feb 10 2008
Subject: [00/19] latency tracer

However, the new tracer is in the rt patchset, starting with patch 2.6.24-rt2:

- cleaned up code
- /debugfs/tracing/ instead of /proc with better control interface
- simultaneous trace of irq off and preempt off
- simultaneous histogram and trace

The examples in this presentation are from the old tracer (mips 2.6.24 + rt patch 2.6.24-rt1).

Tuning, part 1

Instrumentation anomalies led to large reported IRQ off times:

- “interrupts disabled” in idle wait (processor specific)

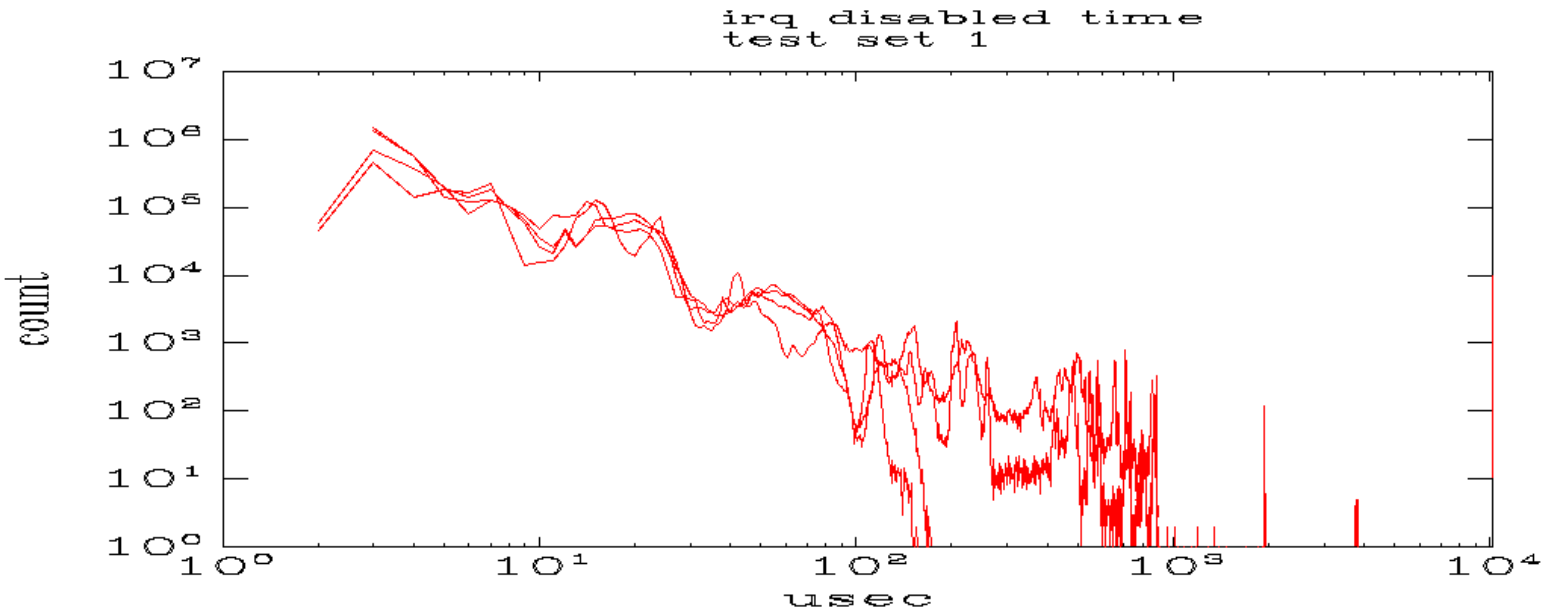
- timer comparison code not handling clock rollover

- timer comparison and capture code not handling switch between raw and non-raw clock sources

The first hint of large latency

/proc/latency_hist/interrupt_off_latency/CPU0

```
#Minimum latency: 2 microseconds.
#Average latency: 27 microseconds.
#Maximum latency: 5725129 microseconds.
#Total samples: 2846758
#There are 3 samples greater or equal than 10240
 microseconds
#usecs                samples
0                           0
1                           0
2                       59063
3                      666520
4                      362079
```



There are many irq disabled periods in the range of 200 – 1000 microseconds.

Multiple tests are shown on this graph.
Each line is the result of a single test run.
A vertical bar on the right edge of the graph indicates one or more measurements beyond the right edge of the graph.

/proc/latency_trace points to a cause

```
latency: 137 us, #10/10, CPU#0 | (M:rt VP:0, KP:0, S
    -----------------
    | task: swapper-0 (uid:0 nice:0 policy:0 rt_prio
    -----------------
 => started at: r4k_wait_irqoff+0x40/0x98 <800213d8>
 => ended at:   irq_exit+0xc0/0xf4 <800567c0>

                  _------=> CPU#
                 / _-----=> irqs-off
                | / _----=> need-resched
                || / _---=> hardirq/softirq
                ||| / _--=> preempt-depth
                |||| /
                |||||     delay
  cmd     pid   ||||| time  |   caller
     \   /      |||||   \   |   /
```

“interrupts disabled” in idle wait

```
switch (c->cputype)
case CPU_TX49XX:
   cpu_wait = r4k_wait_irqoff

static void r4k_wait_irqoff(void)
{
        local_irq_disable();
        if (!need_resched())
                __asm__("wait \n");
        local_irq_enable();
}
```

The WAIT instruction

The WAIT instruction causes the processor to enter a low-power mode. The processor exits from the low-power mode upon an interrupt exception.

So when an interrupt occurs, r4k_wait_irqoff() will immediately re-enable interrupts.

Quick “FIX”

Use the pre-existing MIPS “nowait” boot option

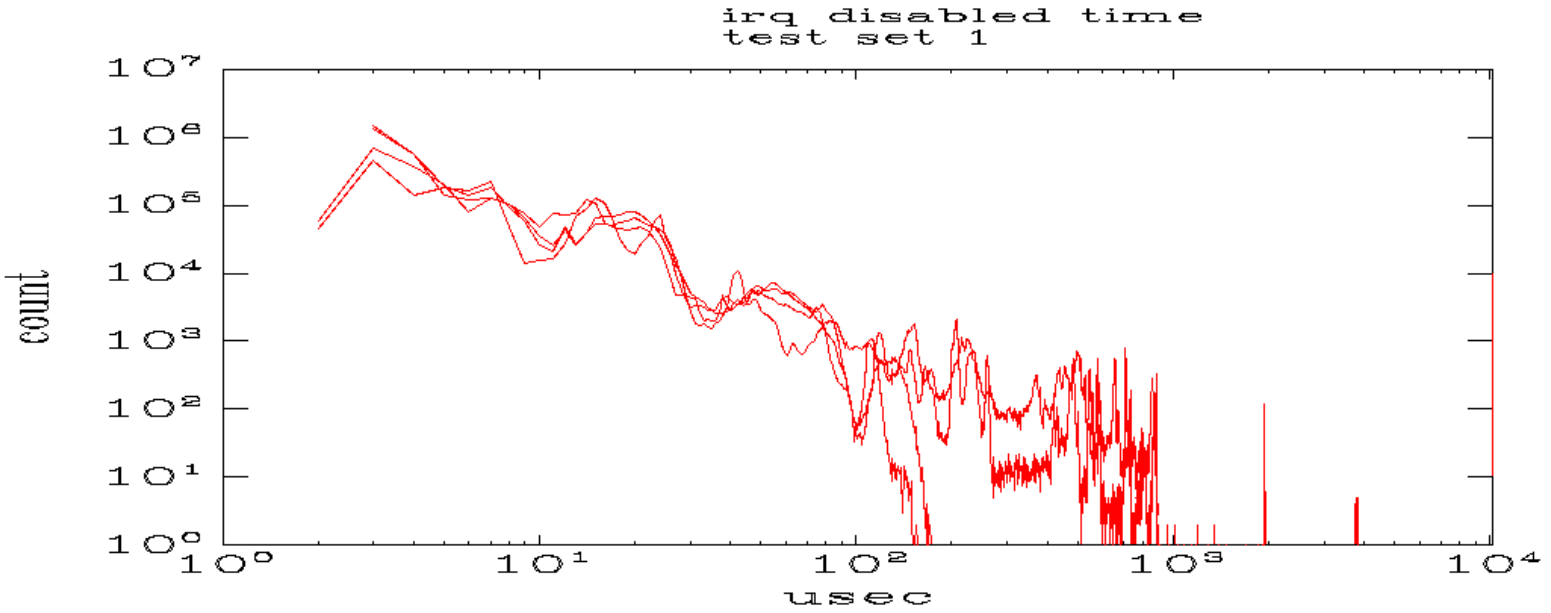
```
static inline void check_wait(void)
{
        if (nowait) {
                printk("Wait instruction disabled.\n");
                return;
        }
        switch (c->cputype)
        case CPU_TX49XX:
           cpu_wait = r4k_wait_irqoff
```

Real “FIX”

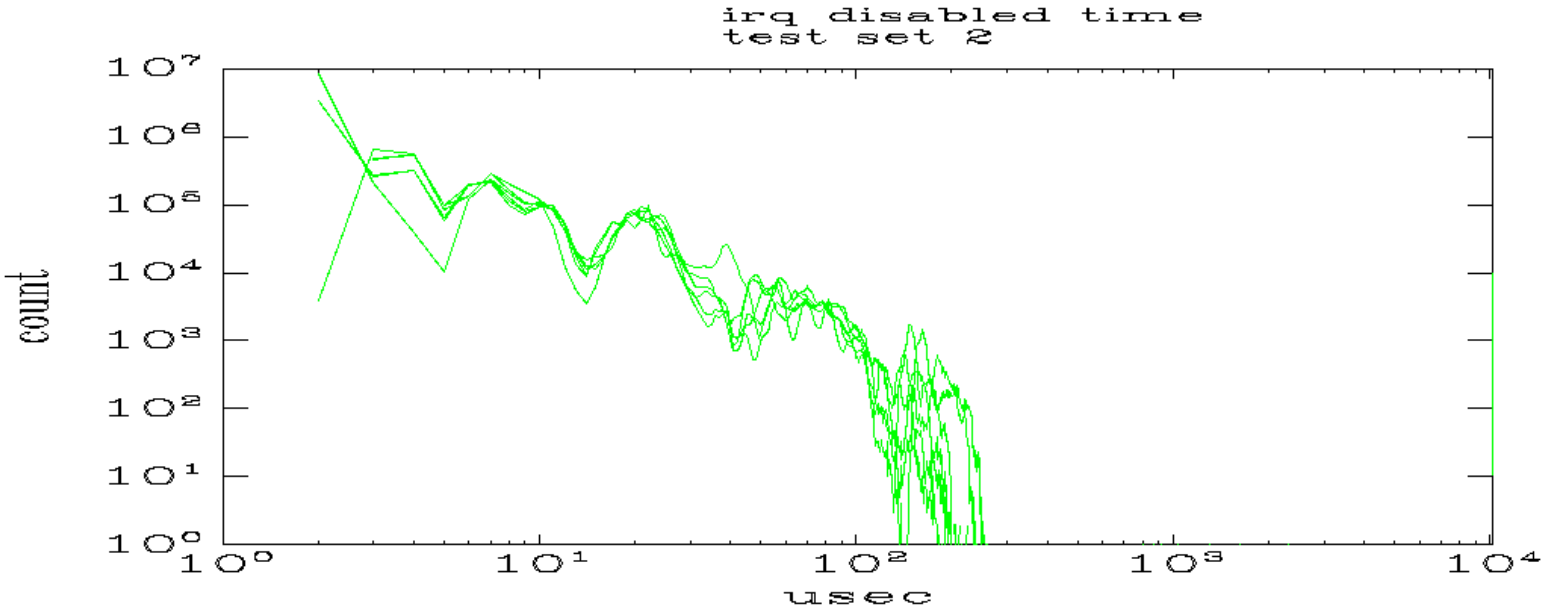
Explicitly stop latency tracing in cpu_idle()

for an example, see

arch/x86/kernel/process_32.c


irq disabled time
test set 1
count
usec
10^7
10^6
10^5
10^4
10^3
10^2
10^1
10^0
10^0
10^1
10^2
10^3
10^4

Fix cpu_wait()


irq disabled time
test set 2
count
usec
10^7
10^6
10^5
10^4
10^3
10^2
10^1
10^0
10^0
10^1
10^2
10^3
10^4

Very large max latency remains

/proc/latency_hist/interrupt_off_latency/CPU0

#Maximum latency: 5725051 microseconds.

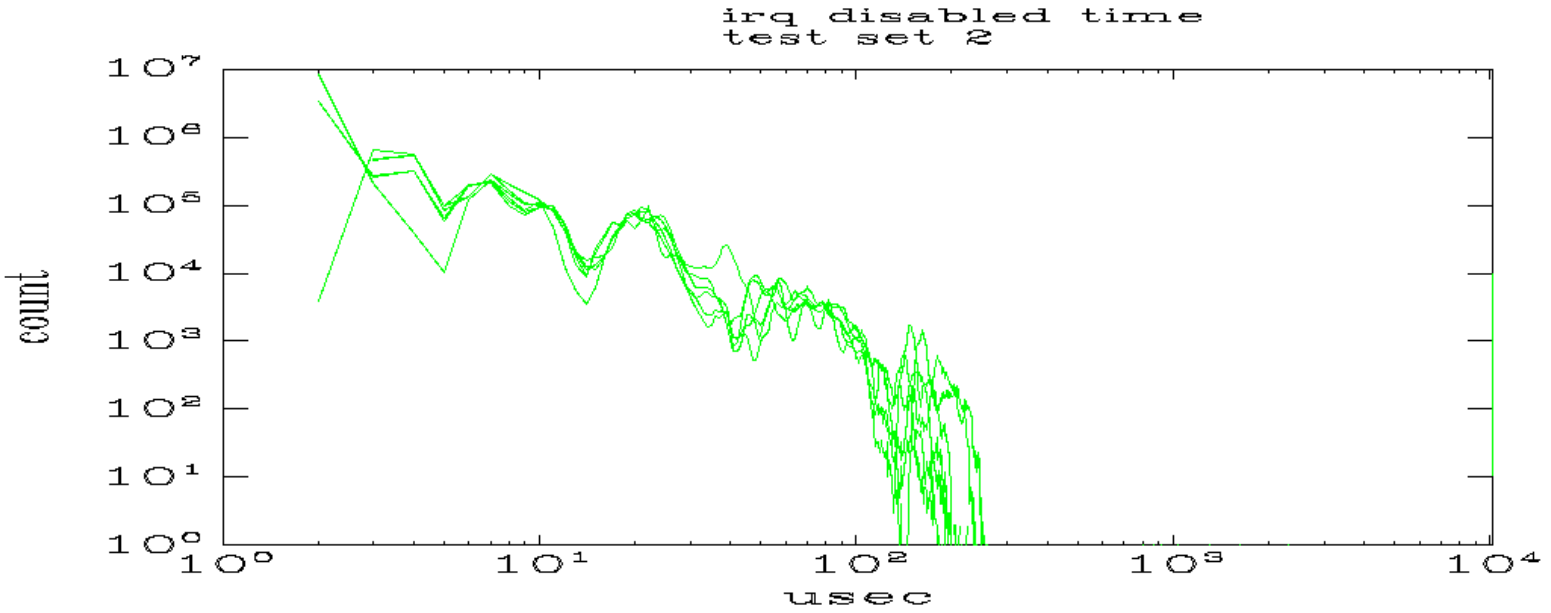
The cause of this was found via normal debugging.

I used the MIPS cycle counter to implement get_cycles(), which the latency tracer uses when the raw cycles mode is enabled. The cycle counter holds a 32 bit value, which rolls over quickly. The latency tracer was not coded to handle timer rollover.

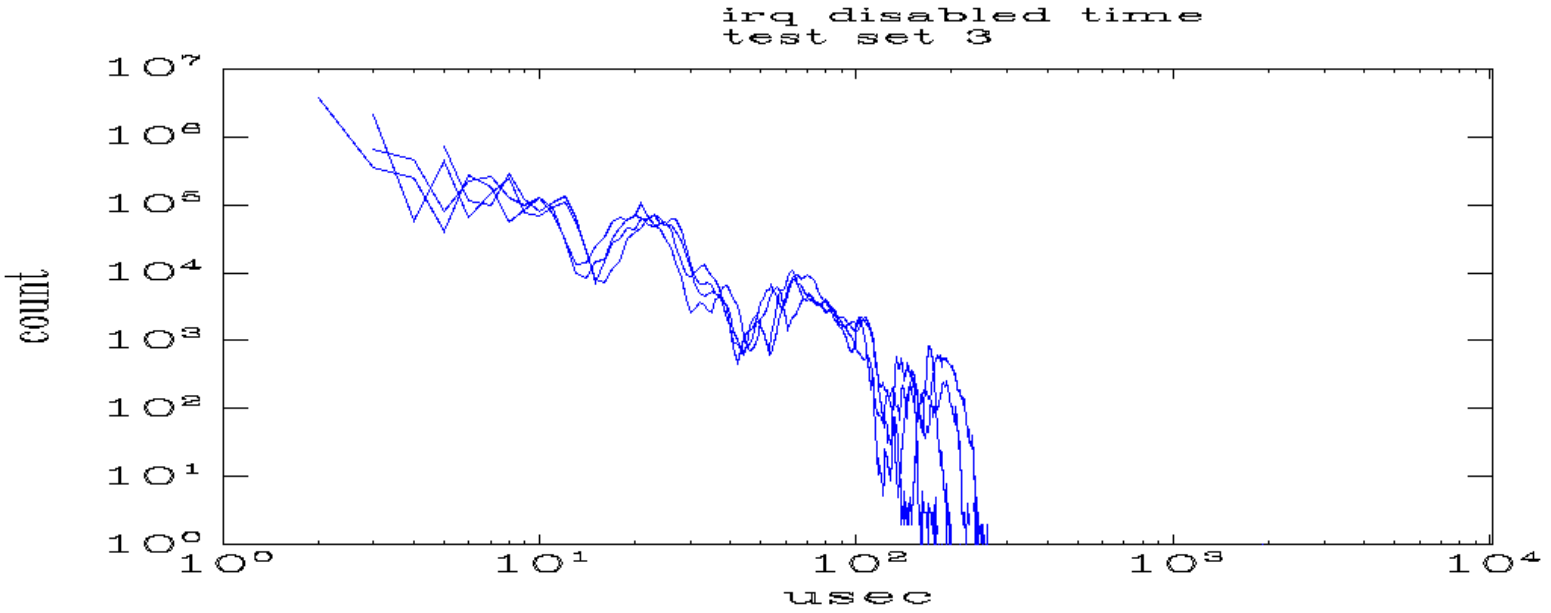
clock rollover FIX

use the same algorithms used for jiffies in
include/linux/jiffies.h

see:
time_after()
time_before()



Fix timestamp compare



Still large max latency remains

/proc/latency_trace reports large latency

/proc/latency_hist/interrupt_off_latency/CPU0

#Maximum latency: 6765 microseconds.

The cause of this was once again found via normal debugging, not through the RT instrumentation.

switch between raw and non-raw clock sources Workaround

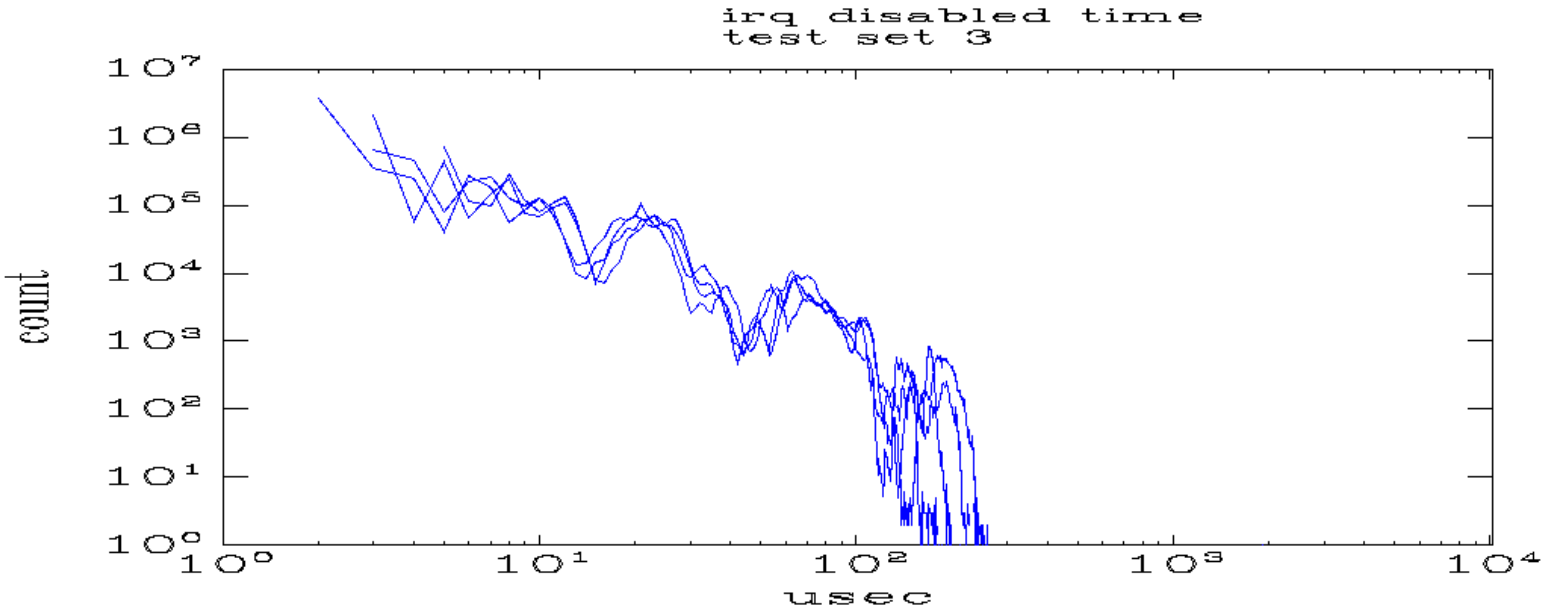
Do not switch back and forth. Use either raw or non-raw for all tracing.

switch between raw and non-raw clock sources FIX

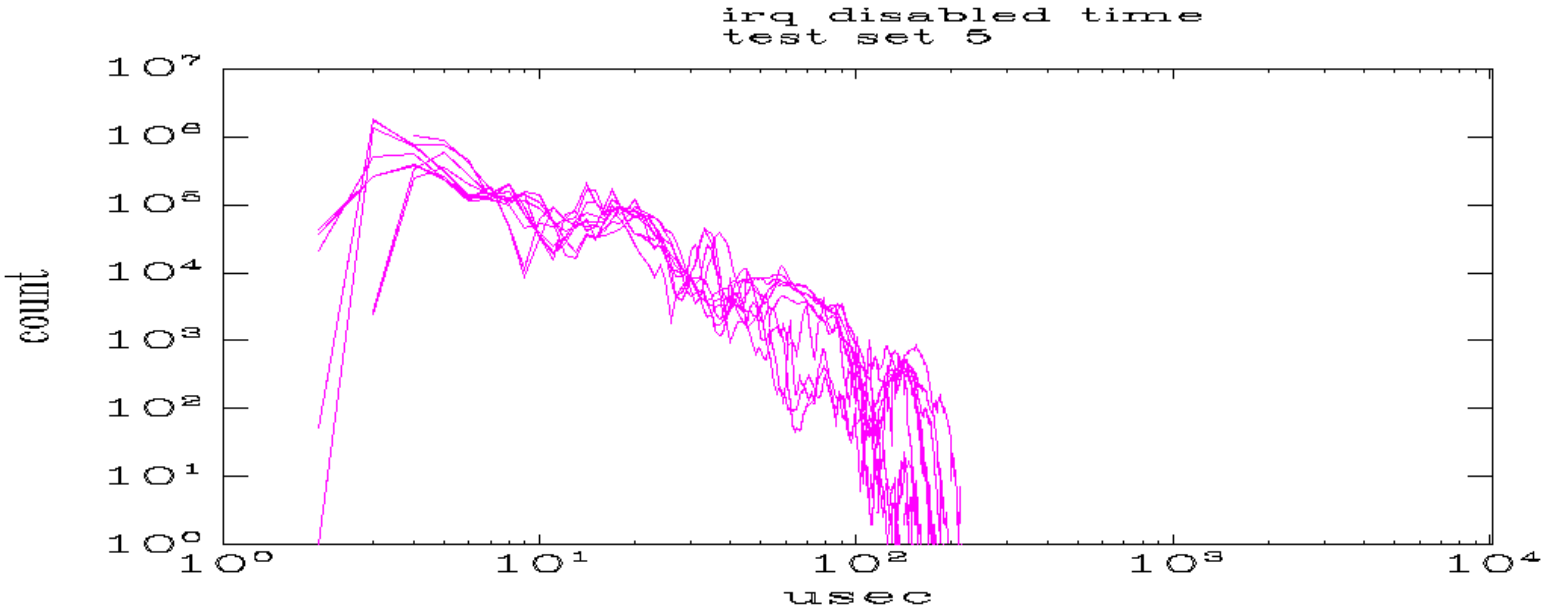
In kernel/latency_trace.c: ____trace()

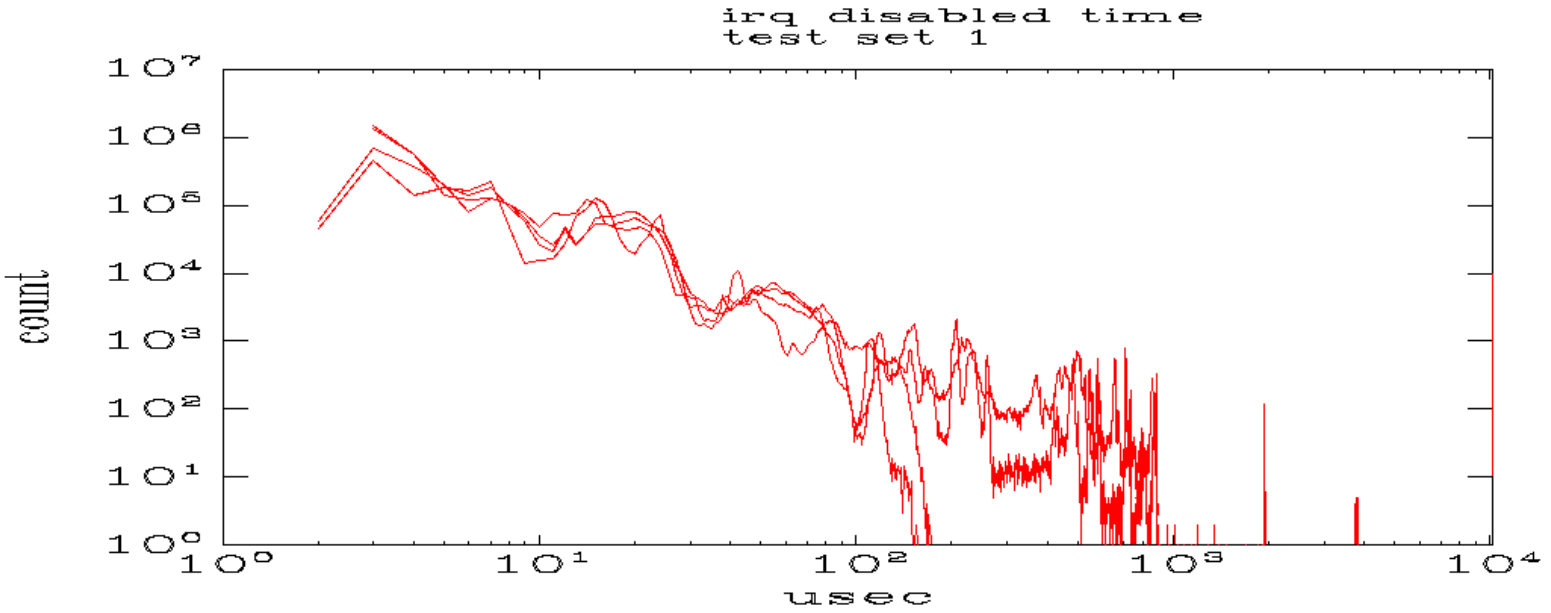
check for switch between raw and non-raw

delete timestamps in other mode from current event

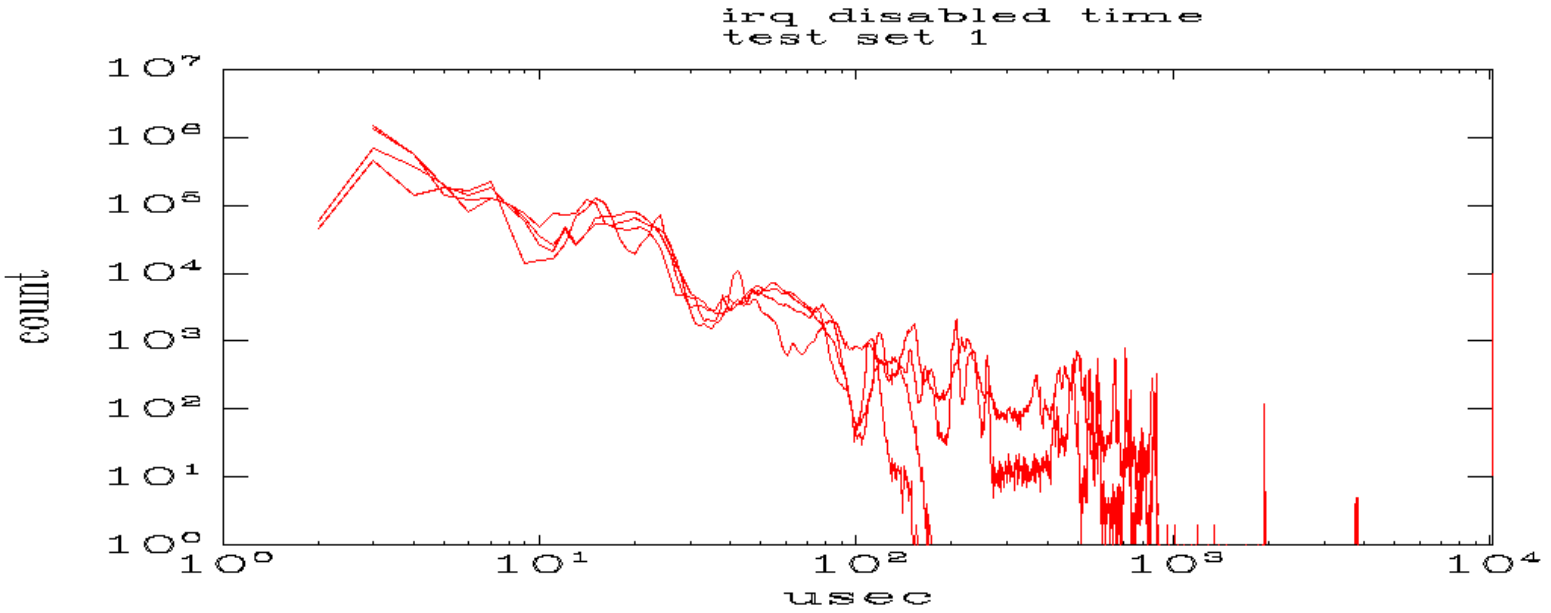


Fix raw / non-raw timestamp transition

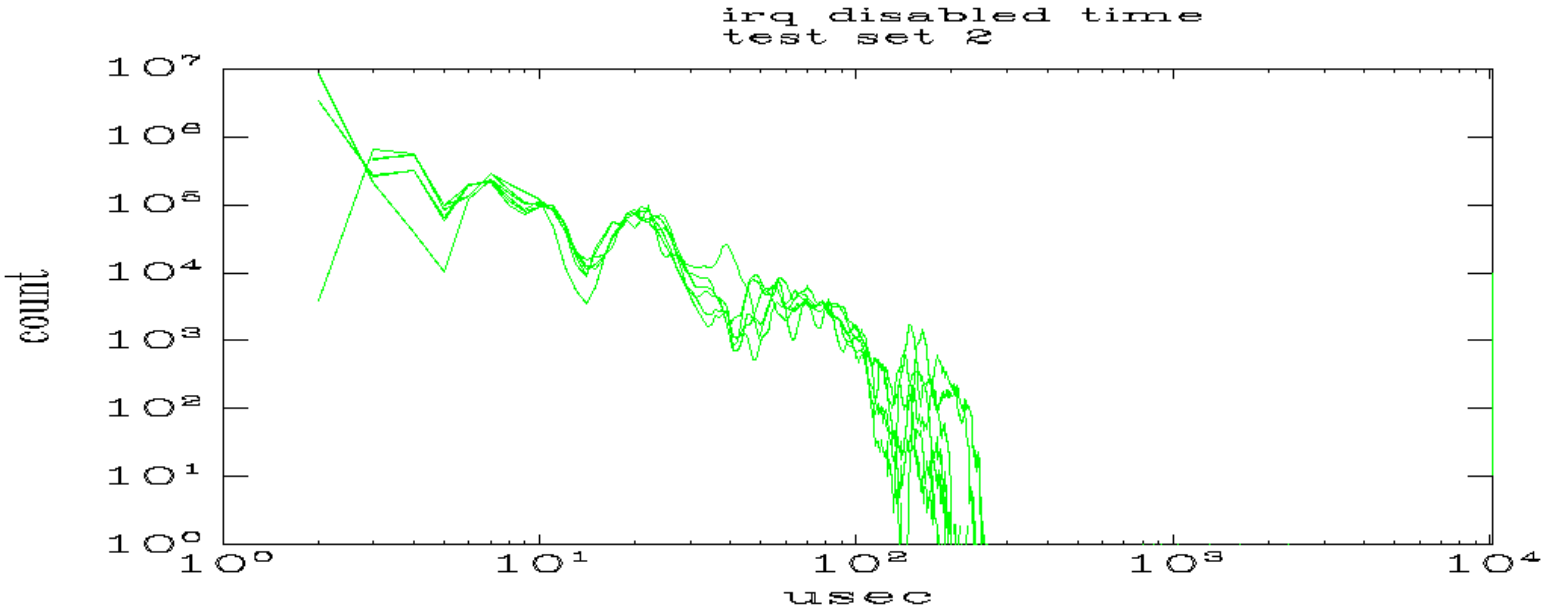


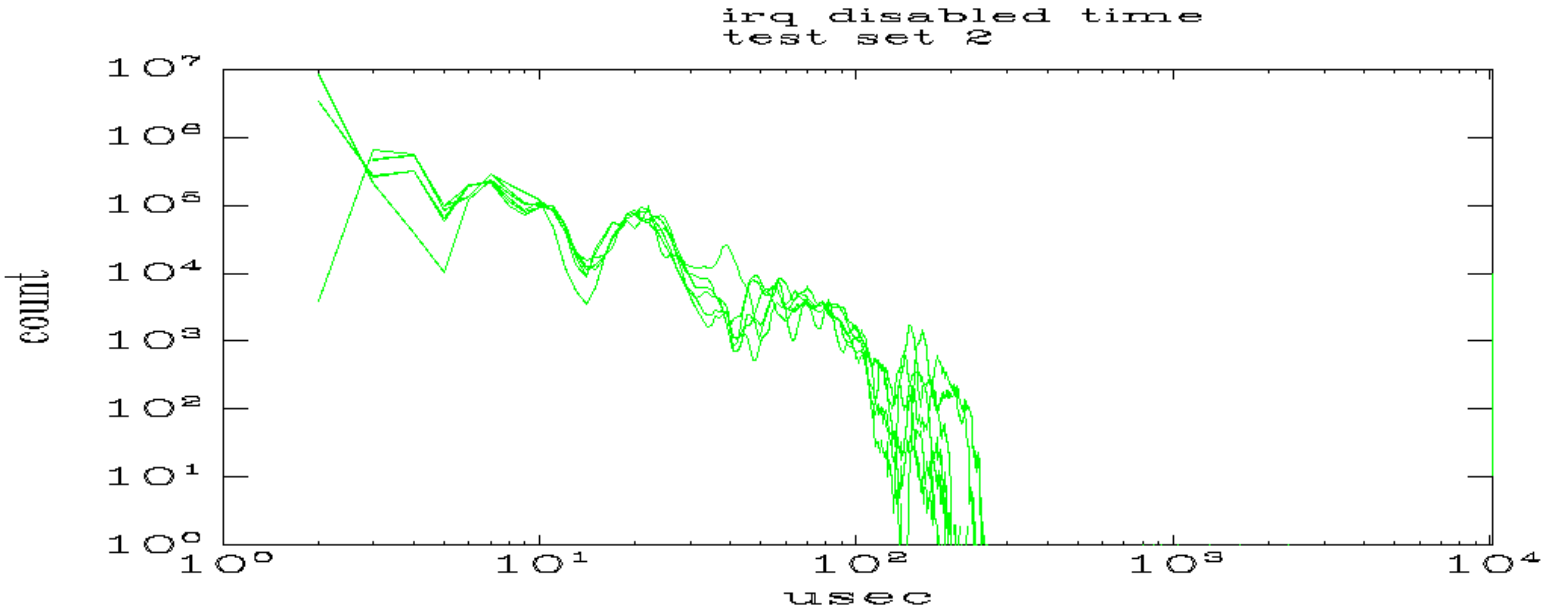


Base measurement

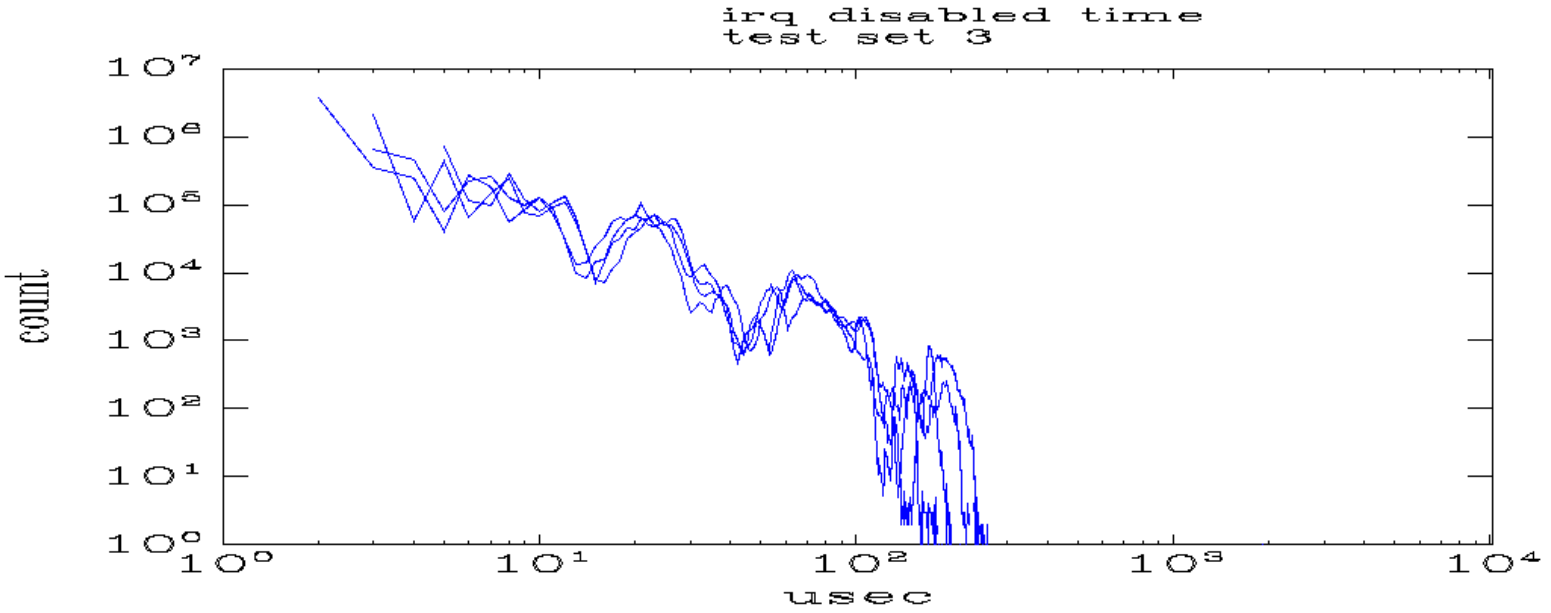

irq disabled time
test set 1
count
usec
10^7
10^6
10^5
10^4
10^3
10^2
10^1
10^0
10^0
10^1
10^2
10^3
10^4

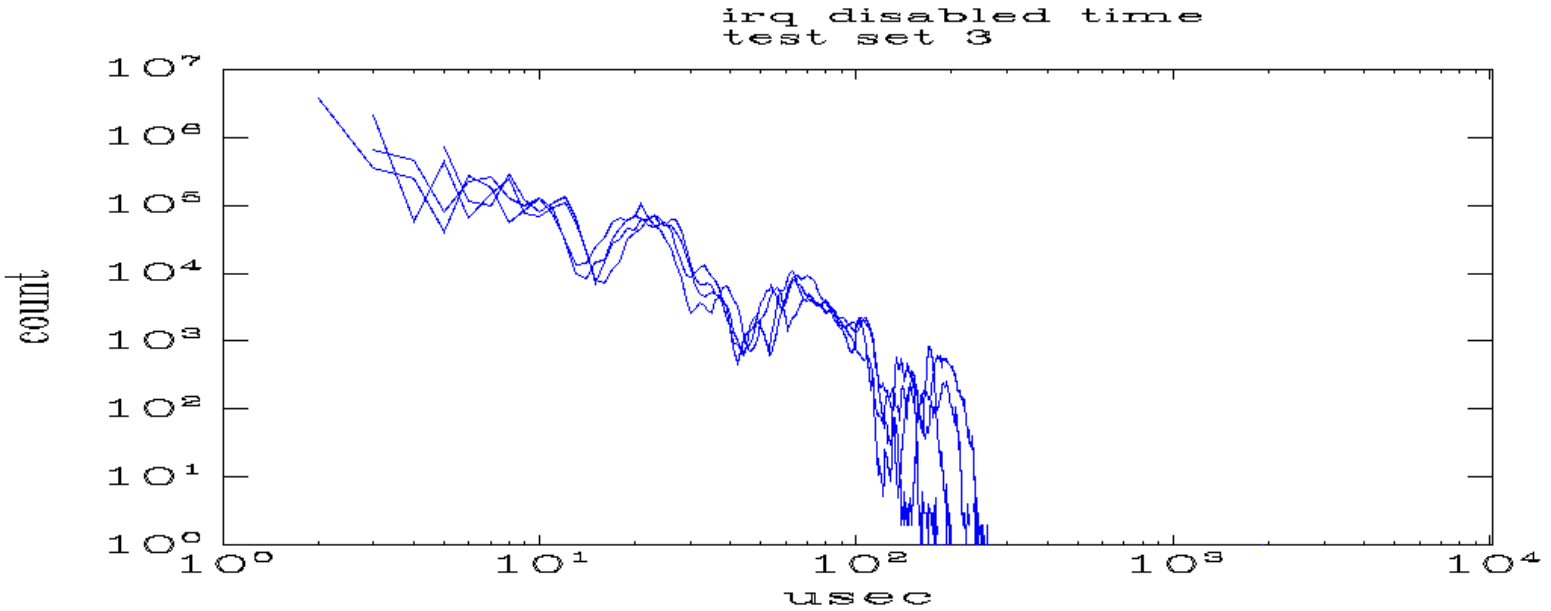
Fix cpu_wait()


irq disabled time
test set 2
count
usec
10^7
10^6
10^5
10^4
10^3
10^2
10^1
10^0
10^0
10^1
10^2
10^3
10^4

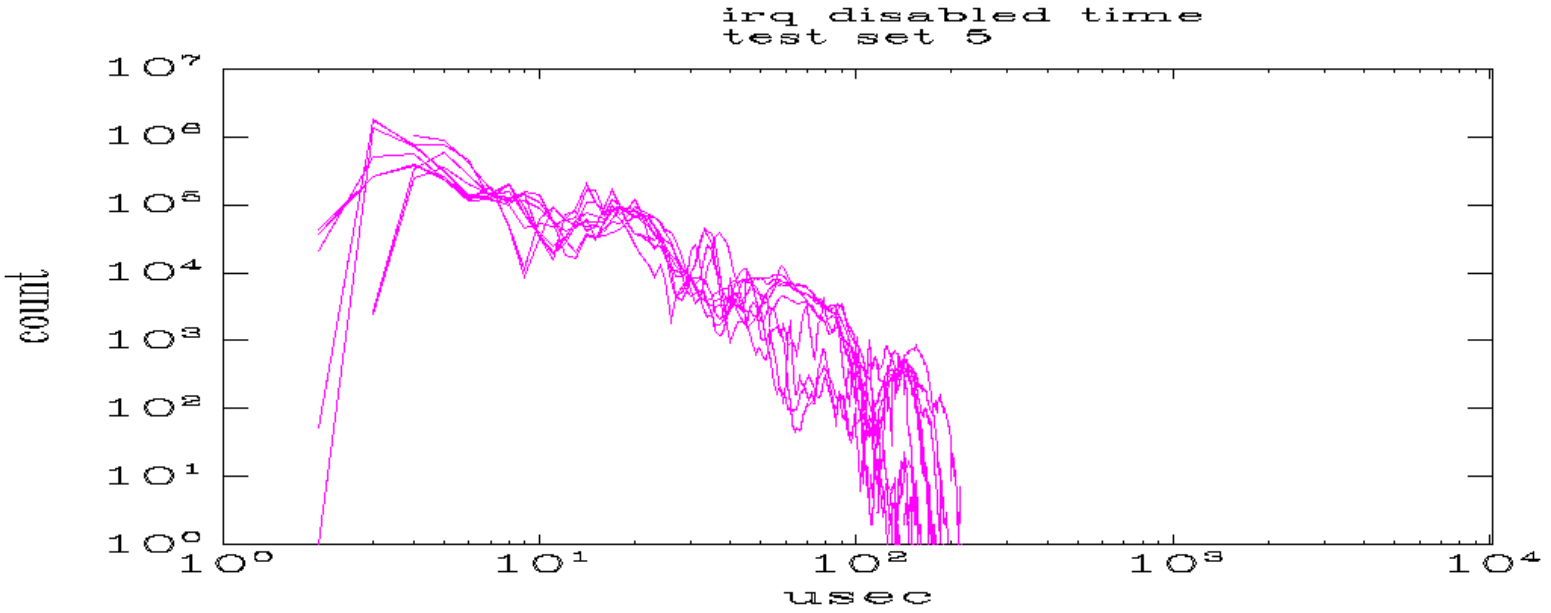


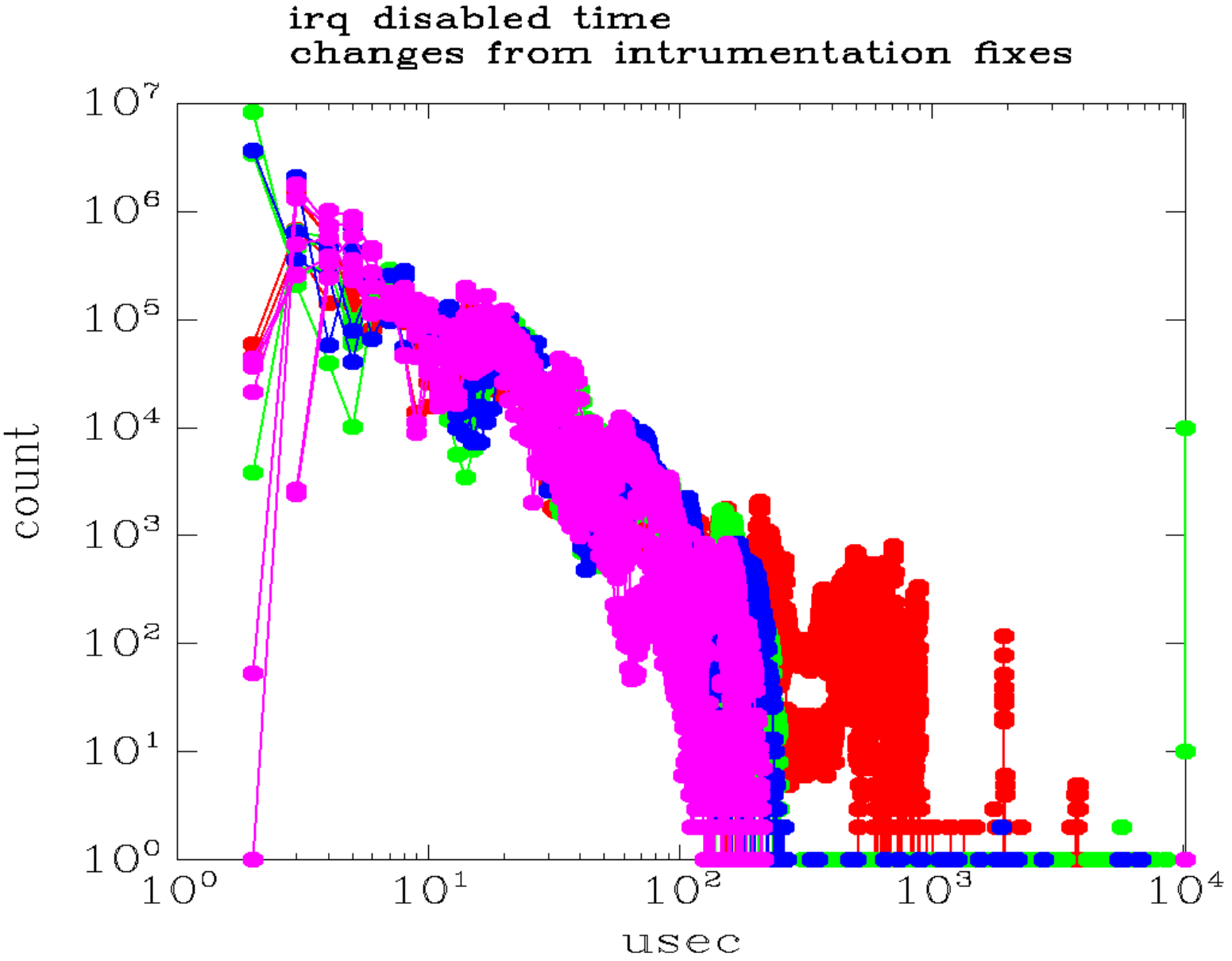
Fix timestamp compare





Fix raw / non-raw timestamp transition




irq disabled time
changes from intrumentation fixes
count
usec
10^0
10^1
10^2
10^3
10^4
10^5
10^6
10^7

The truly outrageous irq disabled times have been resolved, what to do next?

- Fix the thing that disables preemption for the longest time

- Fix the thing that disables interrupts for the longest time

I will stay with irqs disabled for the examples.

/proc/latency_trace revisited

One tool that is useful for examining paths with irqs disabled or preempton disabled is the latency trace. (The previous example included start and end points, but contained no details in between.)

The next several slides show

- the raw output of a latency trace
- the raw output of a latency trace, slightly edited
- hand annotations of the latency trace

```
 latency: 164 us, #22/22, CPU#0 | (M:rt VP:0, KP:0, SP:1 HP:1)
    -----------------
    | task: softirq-timer/0-7 (uid:0 nice:-5 policy:1 rt_prio:50)
    -----------------
 => started at: handle_int+0x10c/0x120 <80021d2c>
 => ended at:   schedule+0xac/0x19c <8025ac70>

                  _------=> CPU#
                 / _-----=> irqs-off
                | / _----=> need-resched
                || / _---=> hardirq/softirq
                ||| / _--=> preempt-depth
                |||| /
                |||||     delay
   cmd     pid |||||  time  |   caller
      \   /    |||||    \   |   /
cyclicte-247   0D...    1us+: handle_int+0x10c/0x120 (<0>)
cyclicte-247   0D.h.   14us+: hrtimer_interrupt+0x9c/0x350 ( 1115 1af13
cyclicte-247   0D.h1   20us+: hrtimer_interrupt+0x164/0x350 ( 1115 1af0
cyclicte-247   0D.h1   49us+: activate_task+0x58/0xa4 <<...>-7> (150 1)
cyclicte-247   0D.h1   53us+: __trace_start_sched_wakeup+0xac/0x19c <<.
cyclicte-247   0DNh1   57us : __trace_start_sched_wakeup+0xac/0x19c <<.
cyclicte-247   0DNh1   58us+: try_to_wake_up+0x1d8/0x1e8 <<...>-7> (150

    (additional entries not shown)
```

Magic Decoder Ring

```
                c
                p
cmdline  pid    uDNHx           sym_name+offset/size of function (data)
-------- ----- ----- -------  ------------------------------------
cyclicte-247   OD.h1   2Ous+: hrtimer_interrupt+Ox164/Ox35O ( 1115 1af1
```

```
  d  irqs off
  D  irqs hard off

  n  need resched delayed
  N  need resched

  H  hardirq && softirq
  h  hardirq
  s  softirq

  x  preempt count as hex

  xxxx!:  time delta to next entry > 100 usec
  xxxx+:  time_delta to next entry >   1 usec
  xxxx :  time_delta to next entry <=  1 usec
```

The entire trace

```
cyclicte-247   OD...    1us+: handle_int+0x10c/0x120 (<0>)
cyclicte-247   OD.h.   14us+: hrtimer_interrupt+0x9c/0x350 ( 1115 1af13
cyclicte-247   OD.h1   20us+: hrtimer_interrupt+0x164/0x350 ( 1115 1af0
cyclicte-247   OD.h1   49us+: activate_task+0x58/0xa4 <<...>-7> (150 1)
cyclicte-247   OD.h1   53us+: __trace_start_sched_wakeup+0xac/0x19c <<.
cyclicte-247   ODNh1   57us : __trace_start_sched_wakeup+0xac/0x19c <<.
cyclicte-247   ODNh1   58us+: try_to_wake_up+0x1d8/0x1e8 <<...>-7> (150
cyclicte-247   ODNh1   87us : activate_task+0x58/0xa4 <<...>-14> (150 2
cyclicte-247   ODNh1   88us : __trace_start_sched_wakeup+0xac/0x19c <<.
cyclicte-247   ODNh1   90us : __trace_start_sched_wakeup+0xac/0x19c <<.
cyclicte-247   ODNh1   91us+: try_to_wake_up+0x1d8/0x1e8 <<...>-14> (150
cyclicte-247   ODNh1   97us+: enqueue_hrtimer+0x4c/0x1b4 ( 1115 1b2e020
cyclicte-247   ODNh.  106us+: clockevents_program_event+0x9c/0x290 ( 111
cyclicte-247   ODNh2  128us : activate_task+0x58/0xa4 <<...>-27> (150 3
cyclicte-247   ODNh2  130us+: __trace_start_sched_wakeup+0xac/0x19c <<.
cyclicte-247   ODNh2  132us : __trace_start_sched_wakeup+0xac/0x19c <<.
cyclicte-247   ODNh2  133us+: try_to_wake_up+0x1d8/0x1e8 <<...>-27> (150
   <...>-7     OD..1  155us+: __schedule+0x3b8/0x638 <cyclicte-247> (0 1
   <...>-7     OD...  160us+: schedule+0xac/0x19c (<0>)
   <...>-7     OD...  162us : trace_hardirqs_on+0xd4/0xf4 (schedule+0xac
```

What are the trace points?

The trace is NOT a list of all functions executed or code paths traversed.

The trace is a collection of interesting locations in various subsystems, which provide a sense of the general flow of control and some related data at some of those locations.

It is up to the analyst to interpolate between those locations (and add additional temporary trace points if needed).

Data Fields

(truncated left part of data to show the right half)

```
OD...  handle_int+0x10c/0x120 (<0>)
OD.h.  hrtimer_interrupt+0x9c/0x350 ( 1115 1af136f1     0)
OD.h1  hrtimer_interrupt+0x164/0x350 ( 1115 1af0f900 80340400)
OD.h1  activate_task+0x58/0xa4 <<...>-7> (150 1)
OD.h1  __trace_start_sched_wakeup+0xac/0x19c <<...>-7> (49 -1)
ODNh1  __trace_start_sched_wakeup+0xac/0x19c <<...>-7> (49 -1)
ODNh1  try_to_wake_up+0x1d8/0x1e8 <<...>-7> (150 0)
ODNh1  activate_task+0x58/0xa4 <<...>-14> (150 2)
ODNh1  __trace_start_sched_wakeup+0xac/0x19c <<...>-14> (49 -1)
ODNh1  __trace_start_sched_wakeup+0xac/0x19c <<...>-14> (49 -1)
ODNh1  try_to_wake_up+0x1d8/0x1e8 <<...>-14> (150 0)
ODNh1  enqueue_hrtimer+0x4c/0x1b4 ( 1115 1b2e0200 80340400)
ODNh.  clockevents_program_event+0x9c/0x290 ( 1115 1af42931 99376)
ODNh2  activate_task+0x58/0xa4 <<...>-27> (150 3)
ODNh2  __trace_start_sched_wakeup+0xac/0x19c <<...>-27> (49 -1)
ODNh2  __trace_start_sched_wakeup+0xac/0x19c <<...>-27> (49 -1)
ODNh2  try_to_wake_up+0x1d8/0x1e8 <<...>-27> (150 0)
OD..1  __schedule+0x3b8/0x638 <cyclicte-247> (0 150)
OD...  schedule+0xac/0x19c (<0>)
OD...  trace_hardirqs_on+0xd4/0xf4 (schedule+0xac/0x19c)
```

Data Fields Hand Annotated

(no Magic Decoder Ring)

```
latency: 164 us, #22/22, CPU#0 | (M:rt VP:0, KP:0, SP:1 HP:1)
 => started at: handle_int+0x10c/0x120 <80021d2c>
 => ended at:   schedule+0xac/0x19c <8025ac70>

handle_int+0x10c/0x120 (<0>)
hrtimer_interrupt+0x9c/0x350 ( 1115 1af136f1     0)          time 1115 1af136f1
hrtimer_interrupt+0x164/0x350 ( 1115 1af0f900 80340400)      time 1115 1af0f900  timer 80340400
activate_task+0x58/0xa4 <<...>-7> (150 1)                    pid   7, PRIO 150, nr_running 1
__trace_start_sched_wakeup+0xac/0x19c <<...>-7> (49 -1)      pid   7, prio 49
__trace_start_sched_wakeup+0xac/0x19c <<...>-7> (49 -1)      pid   7, prio 49
try_to_wake_up+0x1d8/0x1e8 <<...>-7> (150 0)                 pid   7, PRIO 150, PRIO(rq->curr) 0
activate_task+0x58/0xa4 <<...>-14> (150 2)                   pid  14, PRIO 150, nr_running 2
__trace_start_sched_wakeup+0xac/0x19c <<...>-14> (49 -1)     pid  14, prio 49
__trace_start_sched_wakeup+0xac/0x19c <<...>-14> (49 -1)     pid  14, prio 49
try_to_wake_up+0x1d8/0x1e8 <<...>-14> (150 0)                pid  14, PRIO 150, PRIO(rq->curr) 0
enqueue_hrtimer+0x4c/0x1b4 ( 1115 1b2e0200 80340400)         time 1115 1b2e0200  timer 80340400
clockevents_program_event+0x9c/0x290 ( 1115 1af42931 99376)  time 1115 1af42931, delta 99376
activate_task+0x58/0xa4 <<...>-27> (150 3)                   pid  27, PRIO 150, nr_running 3
__trace_start_sched_wakeup+0xac/0x19c <<...>-27> (49 -1)     pid  27, prio 49
__trace_start_sched_wakeup+0xac/0x19c <<...>-27> (49 -1)     pid  27, prio 49
try_to_wake_up+0x1d8/0x1e8 <<...>-27> (150 0)                pid  27, PRIO 150, PRIO(rq->curr) 0
__schedule+0x3b8/0x638 <cyclicte-247> (0 150)                pid 247, PRIO was 0, PRIO is 150
schedule+0xac/0x19c (<0>)
trace_hardirqs_on+0xd4/0xf4 (schedule+0xac/0x19c)
```

Finding source location, data fields (the easy way)

```
OD.h.  hrtimer_interrupt+0x9c/0x350 ( 1115 1af136f1     0)
OD.h1  hrtimer_interrupt+0x164/0x350 ( 1115 1af0f900 80340400)
```

Look at the source, maybe it's obvious (or maybe it's not....).

```
void hrtimer_interrupt(struct clock_event_device *dev)
{
        .....
 retry:
        now = ktime_get();
        hrtimer_trace(now, 0);

        .....

              hrtimer_trace(timer->expires, (unsigned long) timer);
```

Finding source location, data fields

```
(gdb) i line *hrtimer_interrupt+0x9c
Line 1105 of "kernel/hrtimer.c"
   starts at address 0x8006ecf4 <hrtimer_interrupt+156>
   ends at 0x8006ed18 <hrtimer_interrupt+192>.
```

```
1102            now = ktime_get();
1103            hrtimer_trace(now, 0);
1104
1105            expires_next.tv64 = KTIME_MAX;
```

Note that the trace address is the location the trace function returns to.

```
# define hrtimer_trace(a,b)      trace_special((a).tv.sec,(a).tv.nsec,b)
```

```
(gdb) i line *schedule+0xac
Line 43 of "irqflags.h" starts at address 0x8025ac70 <schedule+172>
   and ends at 0x8025ac90 <schedule+204>.

     41 static inline void raw_local_irq_enable(void)
     42 {
     43         __asm__ __volatile__(
                                          "       mfc0    $1,$12 \n”

(gdb) x/i *schedule+0xac
0x8025ac70 <schedule+172>:      mfc0    at,$12

(gdb) x/8i *schedule+0xac - 0x10
0x8025ac60 <schedule+156>:      bnez    v0,0x8025ac38 <schedule+116>
0x8025ac64 <schedule+160>:      nop
0x8025ac68 <schedule+164>:      jal     0x80084c54 <trace_hardirqs_on>
0x8025ac6c <schedule+168>:      nop
0x8025ac70 <schedule+172>:      mfc0    at,$12
0x8025ac74 <schedule+176>:      nop
0x8025ac78 <schedule+180>:      ori     at,at,0x1f
0x8025ac7c <schedule+184>:      xori    at,at,0x1e
```

```
(gdb) i line *schedule+0xac
Line 43 of "irqflags.h" starts at address 0x8025ac70 <schedule+172>
   and ends at 0x8025ac90 <schedule+204>.

(gdb) x/i *schedule+0xac - 0x10
0x8025ac60 <schedule+156>:      bnez    v0,0x8025ac38 <schedule+116>

(gdb) i line *0x8025ac60
Line 3854 of "kernel/sched.c"
   starts at address 0x8025ac60 <schedule+156>
   ends at 0x8025ac68 <schedule+164>.

3852            do {
3853                    __schedule();
3854            } while (unlikely(test_thread_flag(TIF_NEED_RESCHED) ||
3855                              test_thread_flag(TIF_NEED_RESCHED_DELAYED)));
3856
3857            local_irq_enable();
```

Interesting Kernel Paths

1) Timer interrupt

- highres timers code
- wake appropriate threads
- schedule

O(n) algorithm -- more timers expiring at same time will result in a longer maximum IRQ off

Obvious in the latency_trace we were examining.

The entire trace

```
cyclicte-247   OD...    1us+: handle_int+0x10c/0x120 (<0>)
cyclicte-247   OD.h.   14us+: hrtimer_interrupt+0x9c/0x350 ( 1115 1af13
cyclicte-247   OD.h1   20us+: hrtimer_interrupt+0x164/0x350 ( 1115 1af0
cyclicte-247   OD.h1   49us+: activate_task+0x58/0xa4 <<...>-7> (150 1)
cyclicte-247   OD.h1   53us+: __trace_start_sched_wakeup+0xac/0x19c <<.
cyclicte-247   ODNh1   57us : __trace_start_sched_wakeup+0xac/0x19c <<.
cyclicte-247   ODNh1   58us+: try_to_wake_up+0x1d8/0x1e8 <<...>-7> (150
cyclicte-247   ODNh1   87us : activate_task+0x58/0xa4 <<...>-14> (150 2
cyclicte-247   ODNh1   88us : __trace_start_sched_wakeup+0xac/0x19c <<.
cyclicte-247   ODNh1   90us : __trace_start_sched_wakeup+0xac/0x19c <<.
cyclicte-247   ODNh1   91us+: try_to_wake_up+0x1d8/0x1e8 <<...>-14> (15
cyclicte-247   ODNh1   97us+: enqueue_hrtimer+0x4c/0x1b4 ( 1115 1b2e020
cyclicte-247   ODNh.  106us+: clockevents_program_event+0x9c/0x290 ( 11
cyclicte-247   ODNh2  128us : activate_task+0x58/0xa4 <<...>-27> (150 3
cyclicte-247   ODNh2  130us+: __trace_start_sched_wakeup+0xac/0x19c <<.
cyclicte-247   ODNh2  132us : __trace_start_sched_wakeup+0xac/0x19c <<.
cyclicte-247   ODNh2  133us+: try_to_wake_up+0x1d8/0x1e8 <<...>-27> (15
   <...>-7     OD..1  155us+: __schedule+0x3b8/0x638 <cyclicte-247> (0
   <...>-7     OD...  160us+: schedule+0xac/0x19c (<0>)
   <...>-7     OD...  162us : trace_hardirqs_on+0xd4/0xf4 (schedule+0xa
```

Possible Fix

Not investigated yet.

Possible workaround

Avoid large number of timers expiring at the same time.

Interesting Kernel Paths

2) interrupt top half handling followed by preempt_schedule_irq() is a long path with irqs disabled

Found by looking at the intermediate time stamps in a latency trace.

Possible Workaround

Remove or rate limit non-RT interrupts.

In my case, the large interrupt volume is due to network traffic since my root file system is NFS mounted from another host. It is not realistic for any use case that I am expecting.

Possible Fix

```
resume_kernel:
        # THIS IS NOT RECOMMENDED, do not do this unless you really
        # understand the negative effects of enabling irqs here

        raw_local_irq_enable  t0
        raw_local_irq_disable

        lw      t0, kernel_preemption
        beqz    t0, restore_all
        lw      t0, TI_PRE_COUNT($28)
        bnez    t0, restore_all
need_resched:
        <<  code deleted for brevity  >>
        jal     preempt_schedule_irq
```

WARNING

Enabling irqs on the return from interrupts patch allows nested interrupts, which may result in a stack overflow.

If you do not understand the negative effects of allowing nested interrupts, or can not ensure they will not crash your specific system, do not apply this change.

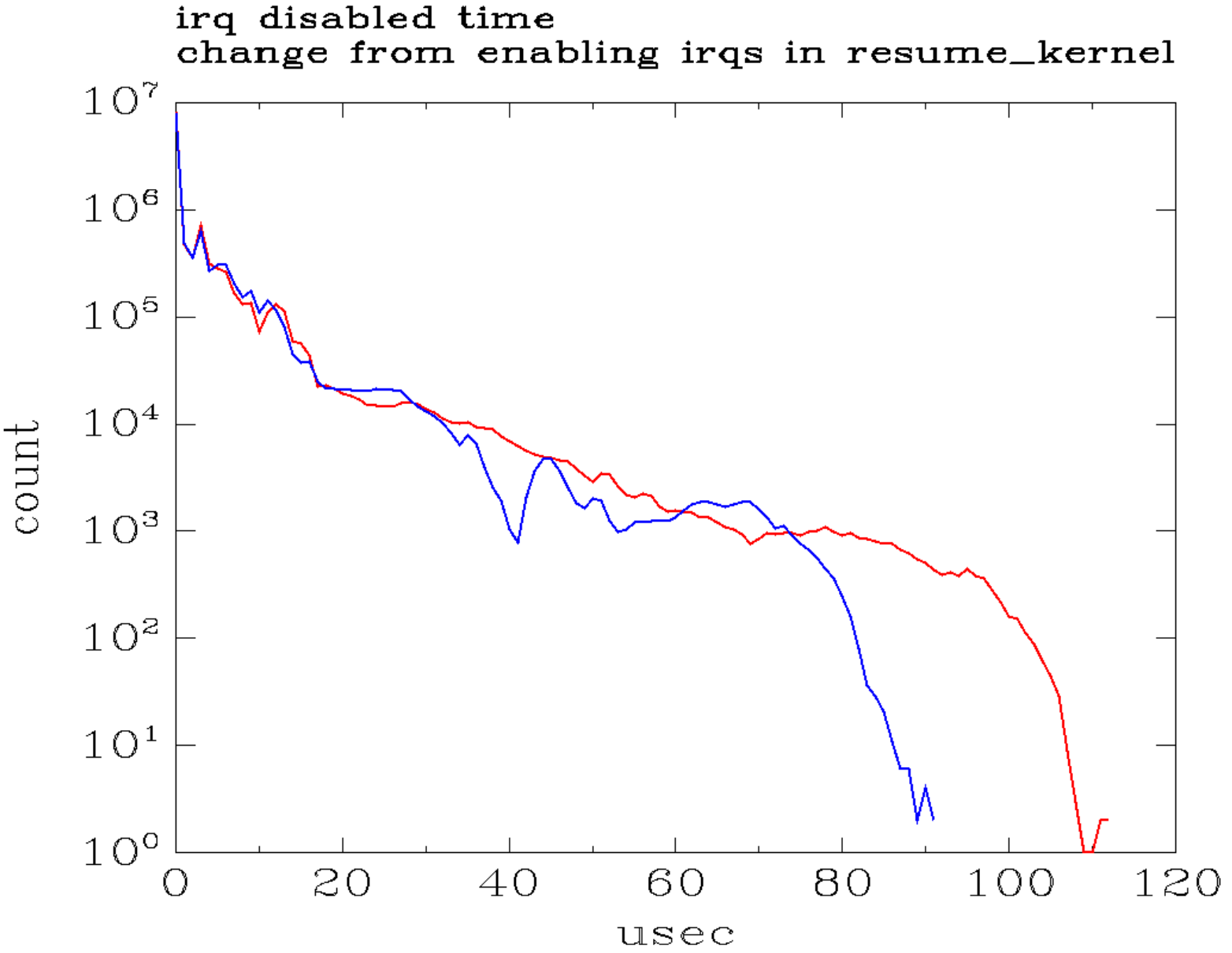
The next three slides are tx49 irq disabled time for two cases:

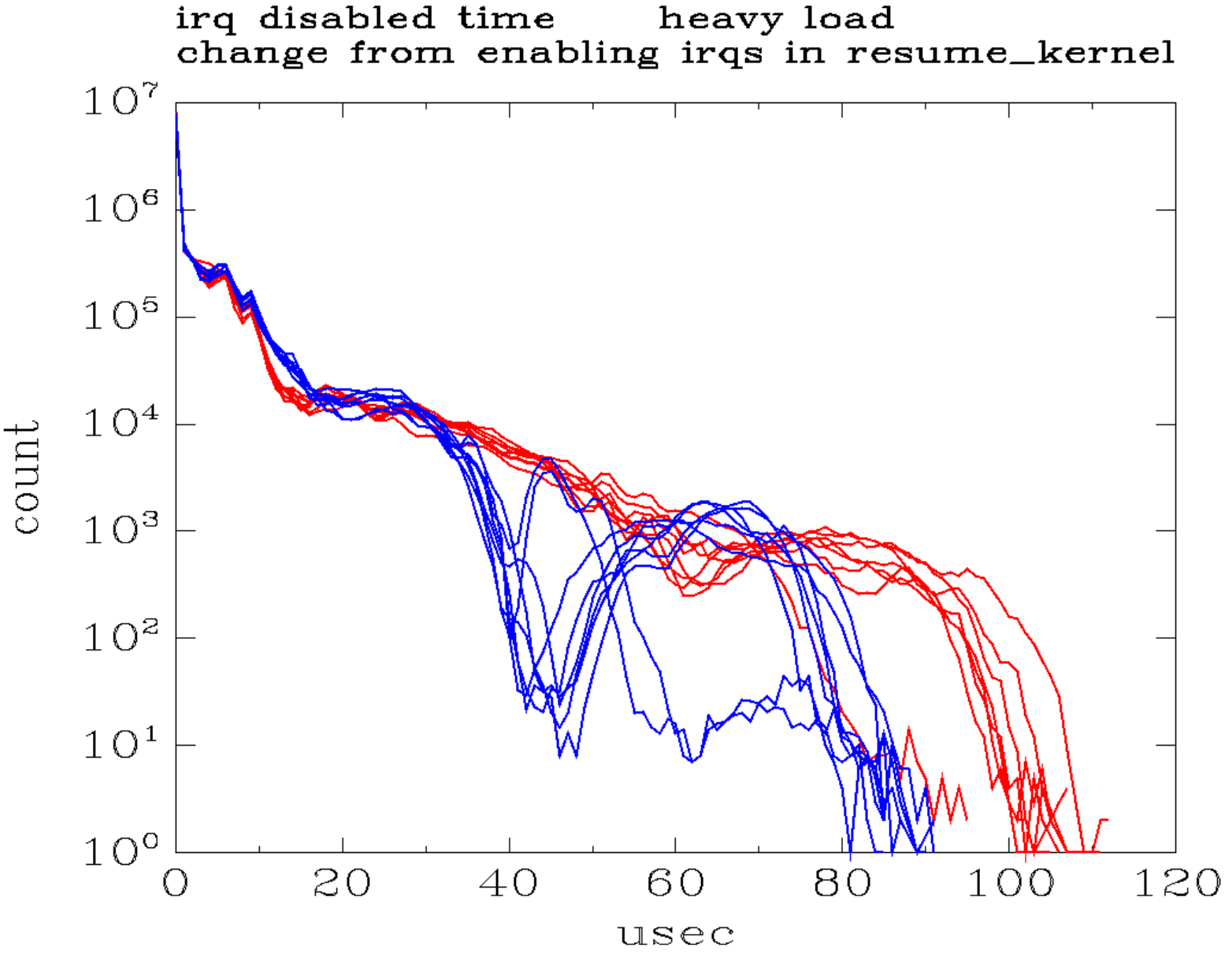
red:
    baseline

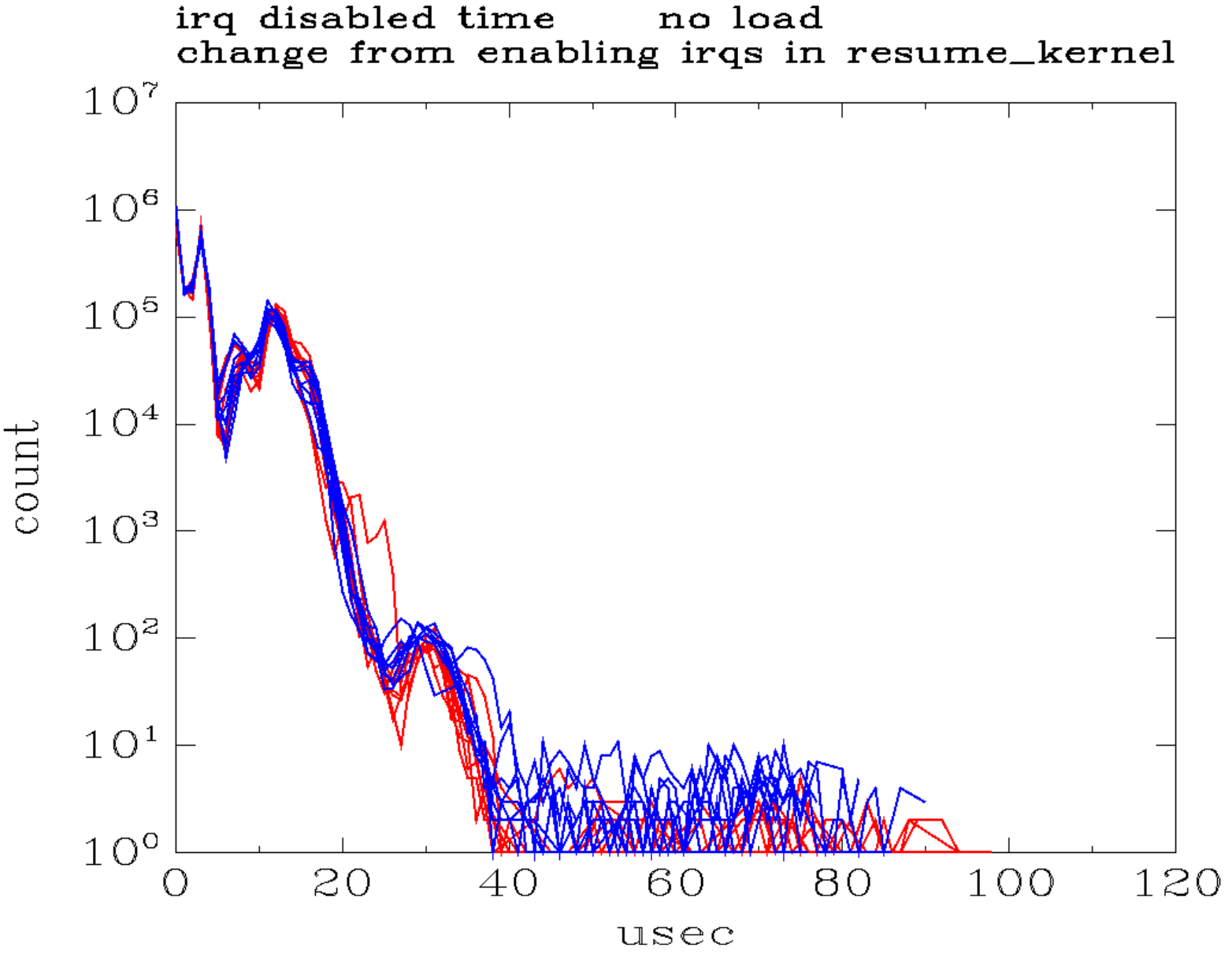
blue:
    Fix for Interesting Kernel Path 3
    resume_kernel

Good improvement for irqs off time (cyclictest RT application results will be seen on a later slide).

(The first slide is the maximum value for each usec for a series of tests.)


irq disabled time
change from enabling irqs in resume_kernel
count
10^7
10^6
10^5
10^4
10^3
10^2
10^1
10^0
0
20
40
60
80
100
120
usec


irq disabled time heavy load
change from enabling irqs in resume_kernel
count
usec
10^7
10^6
10^5
10^4
10^3
10^2
10^1
10^0
0
20
40
60
80
100
120


irq disabled time no load
change from enabling irqs in resume_kernel
count
usec
10^7
10^6
10^5
10^4
10^3
10^2
10^1
10^0
0
20
40
60
80
100
120

```
tx49                250 Mhz    100 usec   (2)
tx49                250 Mhz     86 usec   (2)

tx49                250 Mhz    139 usec   (3)
tx49                250 Mhz    245 usec   (3)

tx49                250 Mhz    112 usec   (4)
tx49                250 Mhz     91 usec   (4)

tx49                250 Mhz    180 usec   (5)
tx49                250 Mhz    269 usec   (5)
```

before resume_kernel fix
with resume_kernel fix, allow only rt irqs in break disable
max irq off (2,4), cyclictest max wakeup latency (3,5)

The moral

Do not lose sight of the most important metric -- meeting the real time application deadline -- while trying to tune the components that cause latency.

The Real Solution

Modify the algorithms and/or data structures to shorten the code path. Probably a major project.

Are these last two fixes reasonable?

No, not really, except in extremely constrained cases. You will never see me submit these simplistic patches upstream. They are not appropriate for a general purpose real time operating system.

Potentially useful if all of the real-time processing is occurring in kernel drivers.

The code shown is much more simplistic than the complete solution that might be reasonable.

Yet another tool

LatencyTOP

Development release 0.1 on Jan 18, 2008

www.latencytop.org

The next tuning tool

Tap into the experts' knowledge -- the web is your friend!

Search engines, wikis, web sites, email lists...

The “Resources” slides at the end of this presentation references some good sources of information.

Normal mail list etiquette applies

1) Search the history
   - The issue and a solution for it may be known.
2) Try to solve the problem yourself
3) Then, consider asking on the list
   - Do your homework, present the data in a concise but complete form.
   - Be prepared to provide additional data and clarifications, collect additional data, and test suggested solutions upon request.
   - Don't expect other people to do your work.
   - Etc...

An example from linux-rt-user

Latencies up to 600us for

- 2.6.24-rc8-rt1: mpc5200 powerpc
- NFS mounted root filesystem
- CONFIG_PREEMPT_RCU_BOOST
  or CONFIG_RCU_TRACE not set
  or CONFIG_RCU_TRACE=m

Start of the thread discussing this is on LKML and linux-rt-user:

Subject: Re: 2.6.24-rc8-rt1
From: Wolfgang Grandegger
Date: Thu Jan 17 2008

Thread subject morphs into:

Re: 2.6.24-rc8-rt1: Strange latencies on mpc5200 powerpc

Some examples of recent history

Softirq processing

Are there any known issues, will they be fixed?

USB subsystem (Isochronous might be OK).

Is it usable for a real-time project?

From the mpc5200 powerpc latency thread:

“It's also my suspicion that the high latencies are related to the RCU usage in the network layer, where it's heavily used.”

Current performance results

Very “preliminary” since tuning has not been completed, but there are some things that can be said.

Kernel version:

MIPS 2.6.24 + patch-2.6.24-rt1 + tx4937 fixes

Source of the data on the following slides

(1) “Realtime capabilities of low-end PowerPC and ARM boards for embedded systems”
Alexander Bauer
9th Real Time Linux Workshop

(2, 3, 4, 5) Frank Rowand, March 2008

***** warning: comparing unlike metrics *****

ppc MCP 5200	266 Mhz	120 usec	(1)
tx49	250 Mhz	90 usec	(2)
tx49	250 Mhz	139 usec	(3)
tx49	250 Mhz	91 usec	(4)
tx49	250 Mhz	180 usec	(5)
arm PXA270	260 Mhz	600 usec	(1)

(1) moderate load (3 cyclictest threads, ping flood)
    metric: cyclictest max wakeup latency
(2,3) light load (5 cyclictest threads)
(4,5) heavy load (5 cyclictest threads, 9 ls -lR)
    0% cpu idle
    (2,4) metric: max IRQ disabled time
    (3,5) metric: cyclictest max wakeup latency

Alexander Bauer Tests

```
ppc MCP 5200
arm PXA270
```

cyclictest -q -n -t 3 -p 99

Background load:

ping flood

Frank Rowand Tests

Toshiba TX4937 Reference Board

cyclictest -p 80 -t 5 -n -l 100000

Background load either:

1) none
2) 9 instances of:
   ls -lR / >/dev/null

/ is nfs mounted from host

Frank Rowand Tests - priorities

```
PID COMMAND          RTPRIO CLS
  3 IRQ-7                50 FF
  4 IRQ-11               50 FF
  5 posix_cpu_timer      99 FF
  6 softirq-high/0       50 FF
  7 softirq-timer/0      50 FF
  8 softirq-net-tx/      50 FF
  9 softirq-net-rx/      50 FF
 10 softirq-block/0      50 FF
 11 softirq-tasklet      50 FF
 12 softirq-sched/0      50 FF
 13 softirq-hrtimer      50 FF
 14 softirq-rcu/0        50 FF
 16 events/0              1 FF
 19 krcupreemptd          1 FF
 24 IRQ-13               50 FF
 26 IRQ-16               50 FF
137 cyclictest            - TS
138 cyclictest            - TS
140 cyclictest           80 FF
142 cyclictest           79 FF
143 cyclictest           78 FF
144 cyclictest           77 FF
145 cyclictest           76 FF
```

Measurement Overhead

Enabling measurement instrumentation adds significant overhead. Remember to disable it before trying to measure actual real time behaviour.

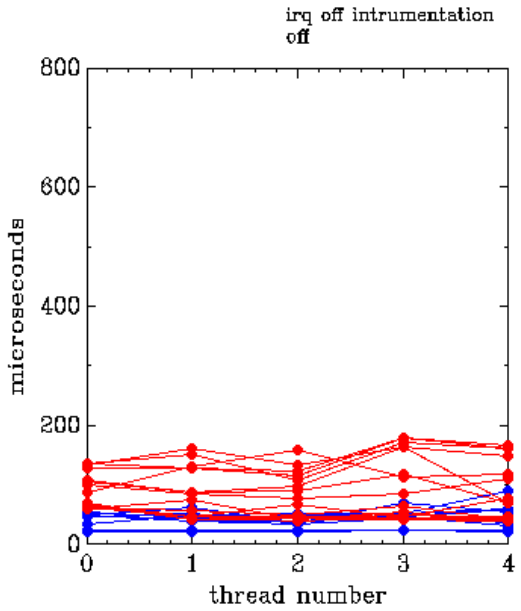
The next slide is tx49 cyclictest latency data (blue is average, red is max), for three cases:

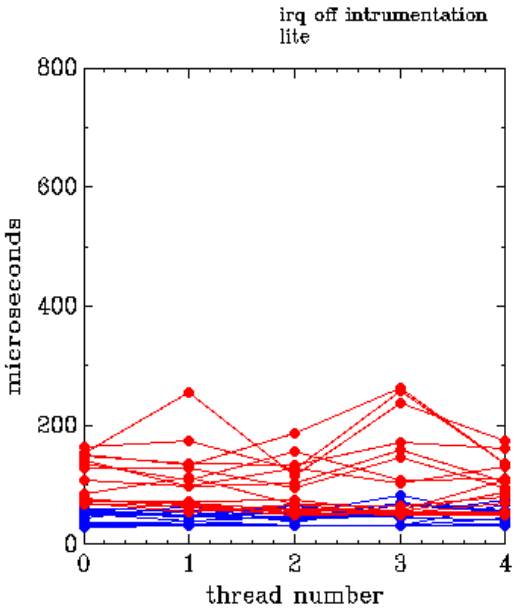
- no latency tracing enabled
- trace-lite enable
- preempt-rt patch latency tracing enabled

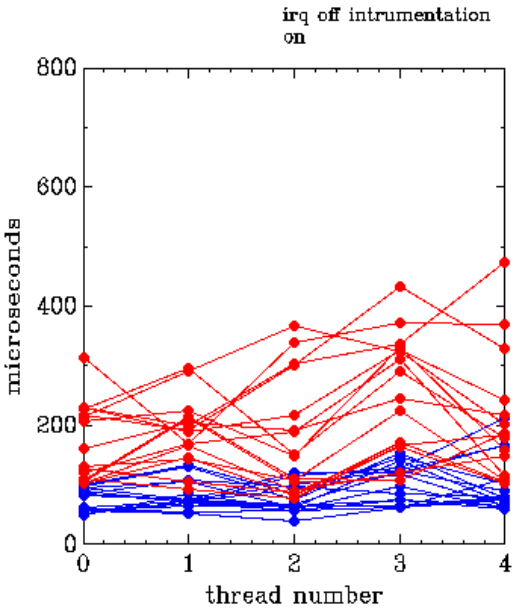
The data shows the large performance impact of enabling latency tracing.

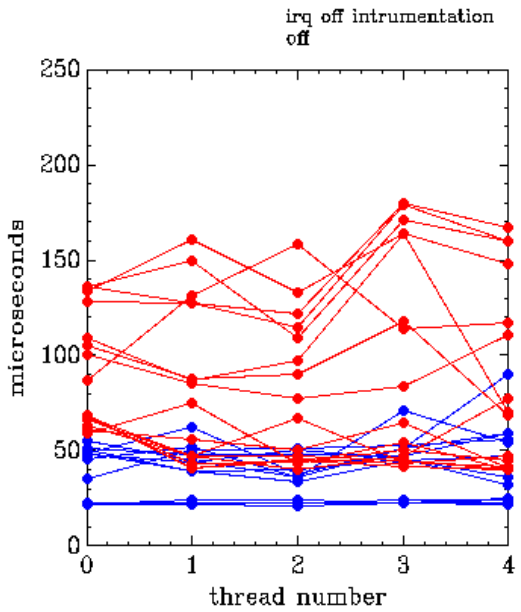
Each individual graph contains multiple lines, where each line is the result of a single test run.

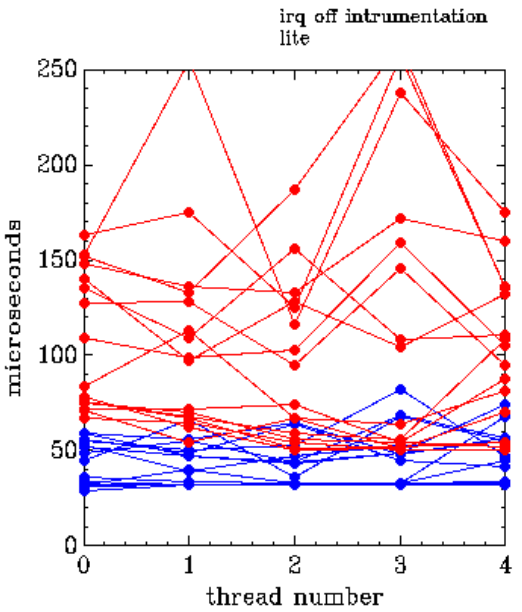
½ of the tests have no “ls” background load.

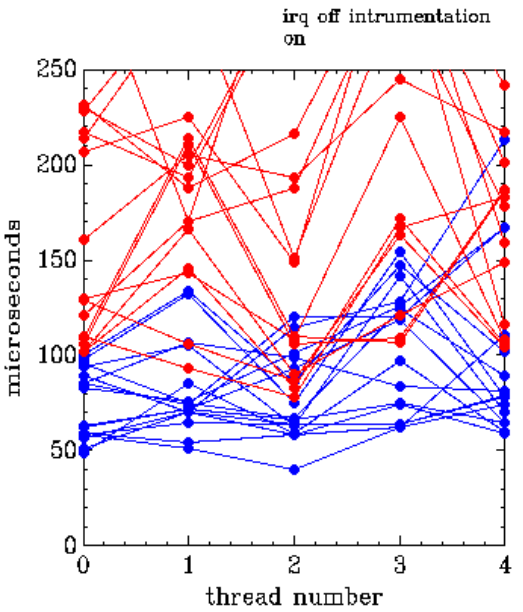

irq off intrumentation
off
microseconds
thread number
0
200
400
600
800
0
1
2
3
4


irq off intrumentation
lite
microseconds
thread number
0
200
400
600
800
0
1
2
3
4


irq off intrumentation
on
microseconds
thread number
0
200
400
600
800
0
1
2
3
4


irq off intrumentation
off
microseconds
thread number
0
50
100
150
200
250
0
1
2
3
4


irq off intrumentation
lite
microseconds
thread number
0
50
100
150
200
250
0
1
2
3
4


irq off intrumentation
on
microseconds
thread number
0
50
100
150
200
250
0
1
2
3
4

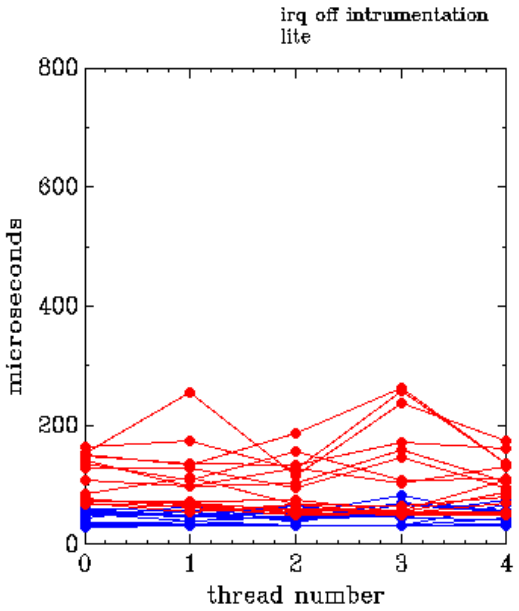
The next slide is tx49 irq disabled time
for three cases:

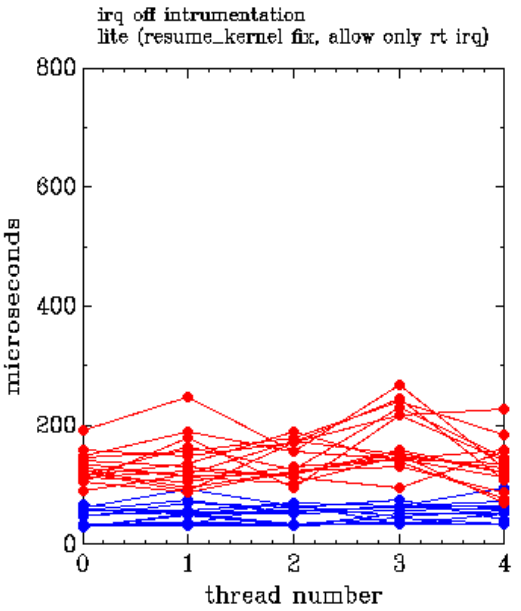
- baseline
- resume_kernel fix, allow only rt irqs in break
- resume_kernel fix

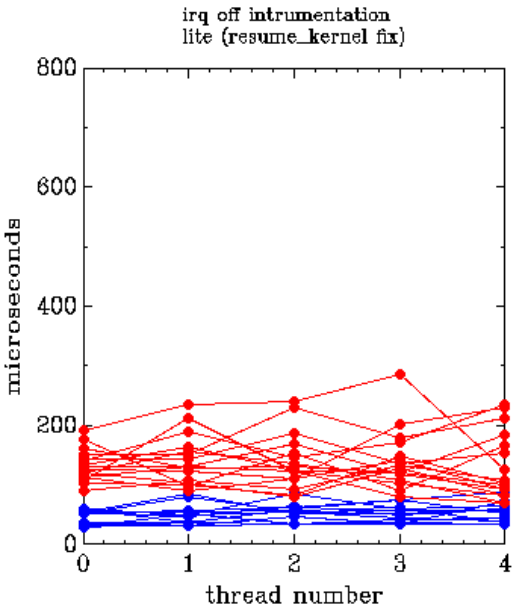
Good improvement was seen earlier for
irqs off time.

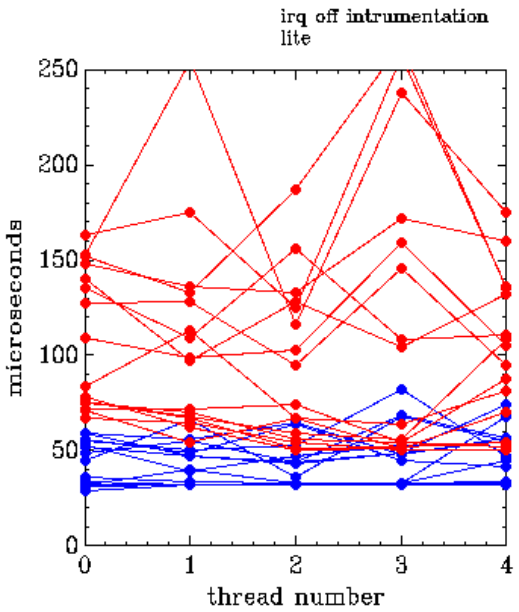
But worst case cyclictest results suffer.

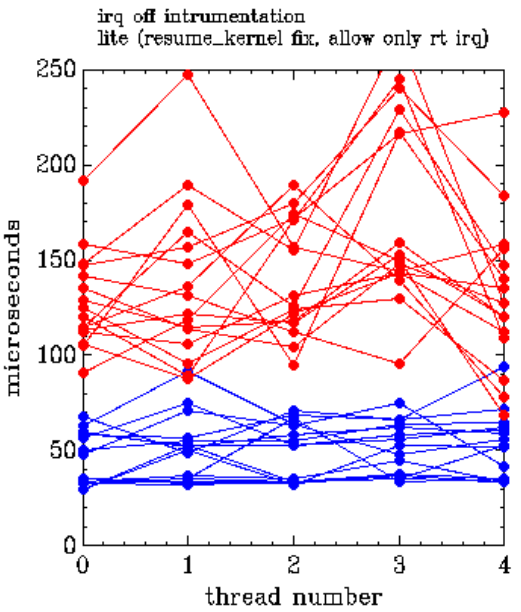
This is an example of how tuning for a single
metric can harm the overall RT application.

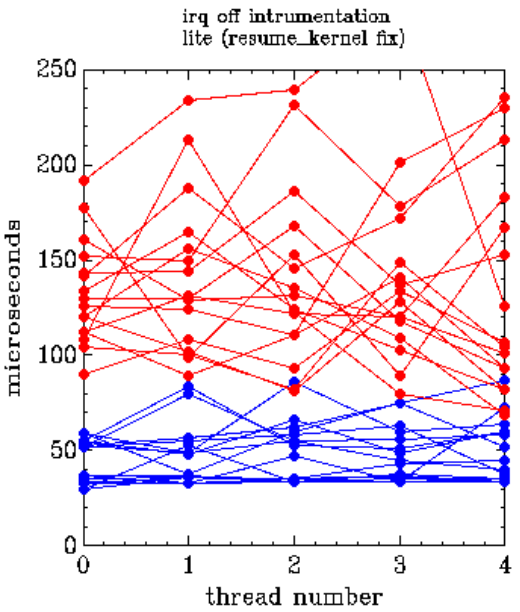

irq off intrumentation
lite
microseconds
thread number
0
200
400
600
800
0
1
2
3
4


irq off intrumentation
lite (resume_kernel fix, allow only rt irq)
microseconds
thread number
0
200
400
600
800
0
1
2
3
4


irq off intrumentation
lite (resume_kernel fix)
microseconds
thread number
0
200
400
600
800
0
1
2
3
4


irq off intrumentation
lite
microseconds
thread number
0
50
100
150
200
250
0
1
2
3
4


irq off intrumentation
lite (resume_kernel fix, allow only rt irq)
microseconds
thread number
0
50
100
150
200
250
0
1
2
3
4


irq off intrumentation
lite (resume_kernel fix)
microseconds
thread number
0
50
100
150
200
250
0
1
2
3
4

The next slides are tx49 interrupts disabled time for two cases:

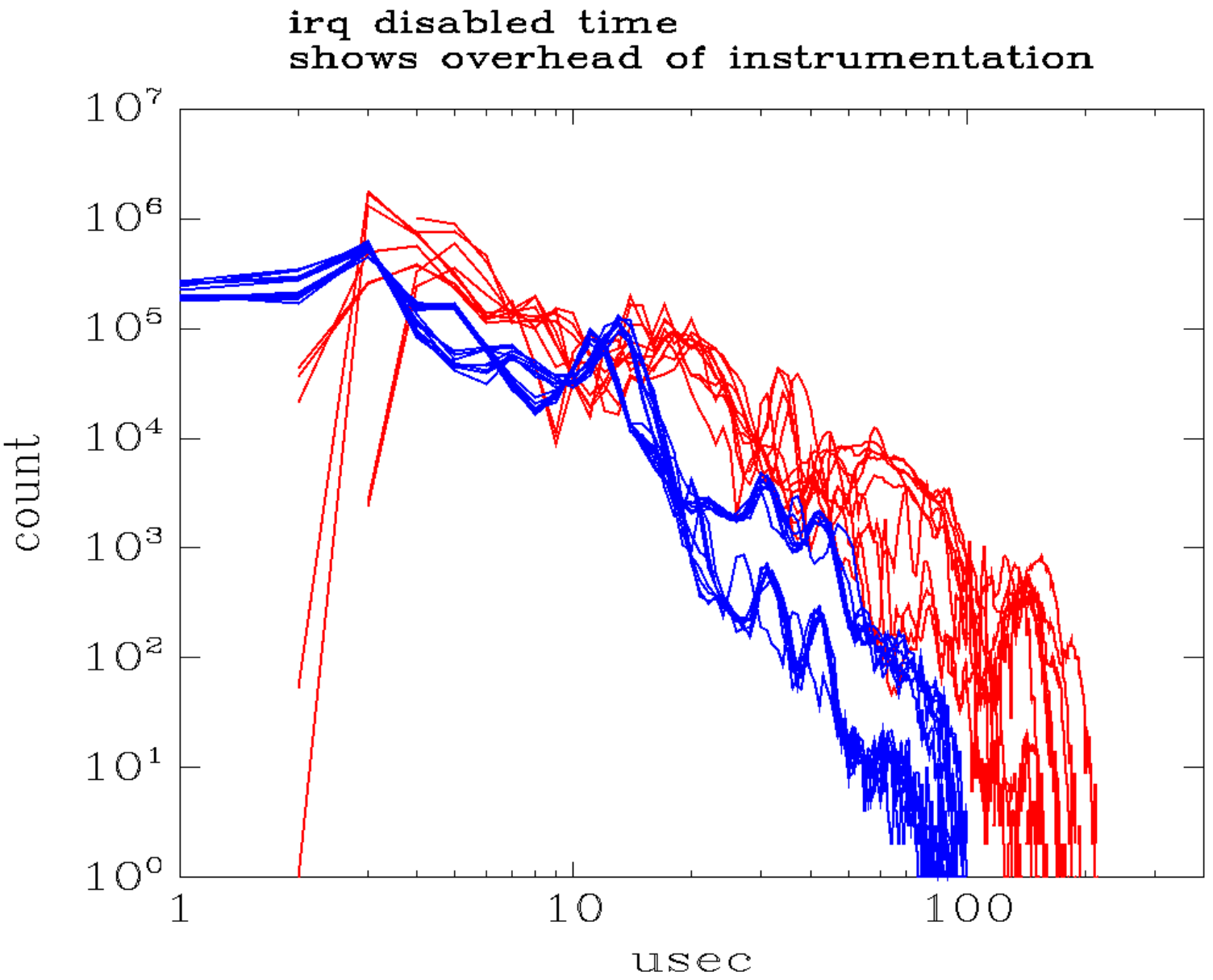
red:

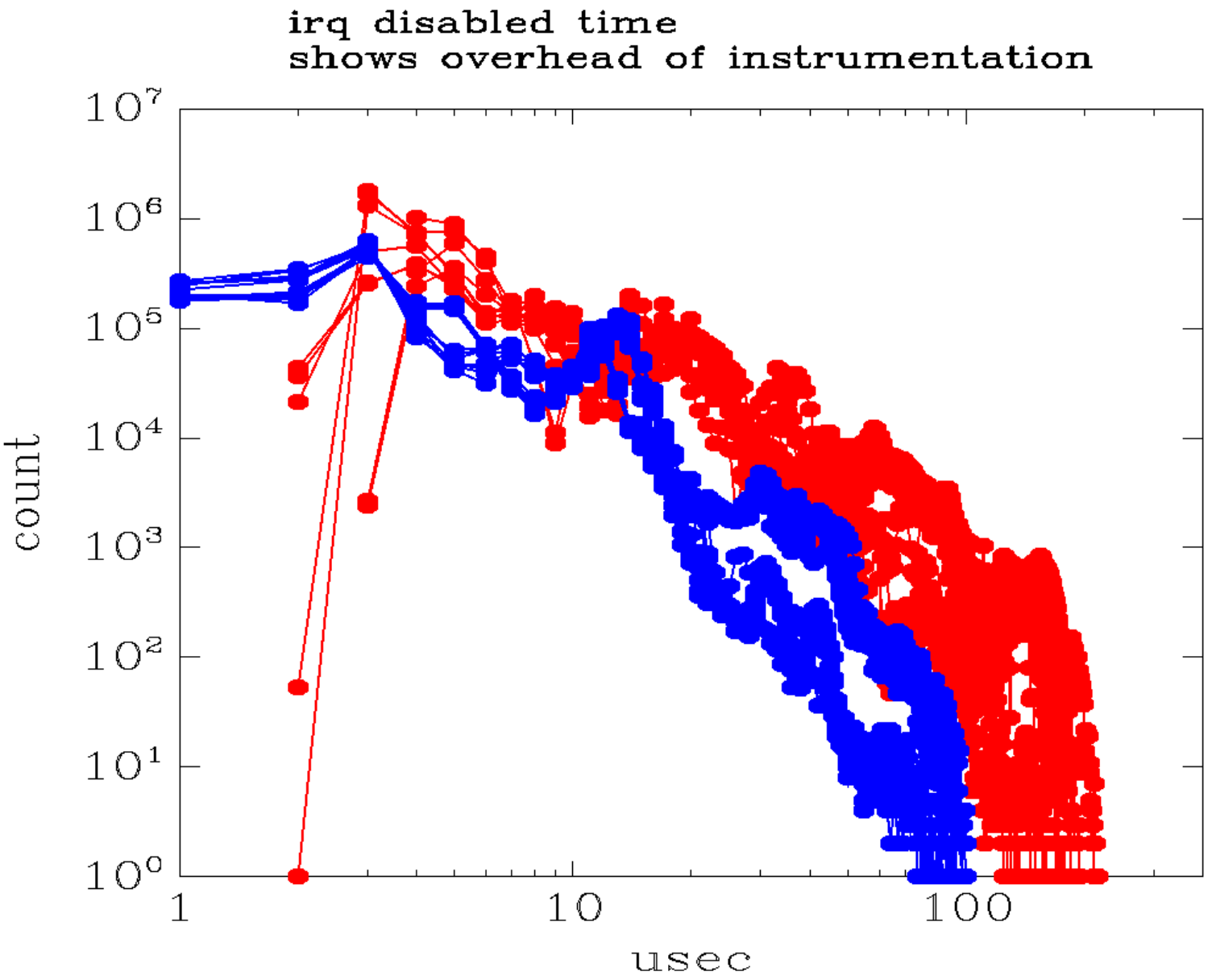
preempt-rt patch latency tracing enabled

blue:

trace-lite enabled

The data again shows the large performance impact of enabling latency tracing.


irq disabled time
shows overhead of instrumentation
count
usec
10^7
10^6
10^5
10^4
10^3
10^2
10^1
10^0
1
10
100


irq disabled time
shows overhead of instrumentation
count
usec
10^7
10^6
10^5
10^4
10^3
10^2
10^1
10^0
1
10
100

lmbench “results”

The overhead for non-basic operations on RT-PREEMPT varies from small to moderate to large.

A few cases where the rt-preempt overhead is smaller than for SMP kernel on UP hardware.

There are inconsistencies between test runs (3 runs per configuration) which implies the data is not reliable.

No attempt was made to validate this data.

Is this lmbench data valid?

It should be viewed with suspicion.

No attempt was made to validate the results (this was just a quick attempt to collect a few data points to see if they would provide some insights into the overhead of preempt-rt).

Not consistent with other reports of recent versions of the kernel on x86, such as Siro Arthur, et al, 9th Real Time Linux Workshop.

Is this lmbench data valid?

Siro Arthur, et al

“Apparently PREEMPT RT has no significant degrading impact on the general performance of the system in its current version”
(2.6.21.5, 2.6.22.1, 2.6.23-rc1)

[for] ”...tests against previous version[s] e.g. 2.6.14-rt20 ... the performance of these kernels were significantly below that of the unpatched versions”
(apologies to Siro for severely mangling this quote)

Limitations of the test methodology

- Non-representative workload.
- No attempt to exercise all areas of system functionality.
- Extremely short test duration.

Bottom line:

This is just the start of this specific tuning project.

The new tracer is in the rt patchset, starting with patch 2.6.24-rt2

old file name	new file name
/proc/latency_trace	/debugfs/tracing/latency_trace /debugfs/tracing/trace
/proc/latency_hist/interrupt_off_latency/CPU0	/debugfs/tracing/
/proc/latency_hist/preempt_off_latency/CPU0	/debugfs/tracing/
/proc/latency_hist/wakeup_latency/CPU0	/debugfs/tracing/

see the patch headers for more documentation of control files

LKML:

From: Ingo Molnar
Date: Sun Feb 10 2008
Subject: [10/19] ftrace: add basic support for gcc profiler instrumentation
Subject: [11/19] ftrace: latency tracer infrastructure for documentation
Subject: [12/19] ftrace: function tracer
Subject: [13/19] ftrace: add tracing of context switches
Subject: [14/19] ftrace: tracer for scheduler wakeup latency
Subject: [15/19] ftrace: trace irq disabled critical timings
Subject: [16/19] ftrace: trace preempt off critical timings

Resources

Rtiwiki
  rt.wiki.kernel.org/index.php/Main_Page

rt-user-list
  dir.gmane.org/gmane.linux.rt.user

eLinux.org
  elinux.org/Real_Time

cyclictest
  http://git.kernel.org/?p=linux/kernel/git/tglx/rt-tests.git;a=summary

hackbench
  http://devresources.linux-foundation.org/craiger/hackbench/

LatencyTOP www.latencytop.org

Resources

“Stress actions - things that will kill realtime performance”
and information about test programs and testing
  elinux.org/Realtime_Testing_Best_Practices

A realtime preemption overview
  lwn.net/Articles/146861

What's in the realtime tree
  lwn.net/Articles/252716

Ninth Real-Time Linux Workshop 2007
  lwn.net/Articles/260118
  linuxdevices.com/articles/AT4991083271.html