



Identifying Embedded Real-Time Latency Issues: I-cache and Locks

Finding and fixing the largest causes of latency in a real-time Linux system is a somewhat well known craft. Finding the last 10% of the causes of excessive latency can be a black art. This talk explores the black art, providing insights into the impact of I-cache and locks on real-time latency on an SMP ARM embedded system. As a result of this talk, the audience will be able to identify some signatures of I-cache and lock issues and will have learned some alternative approaches to investigate latency.

This talk is accessible to all technical levels. Those with a basic knowledge of computer architecture will understand the information. Experts will gain valuable insights to add to their toolkit.

Frank Rowand, Sony Network Entertainment

April 11, 2011

110408_1605

Overview

I am going to describe part of an investigation of a relatively small code path that contributes significantly to the maximum **interrupts disabled duration** for some of my test systems.

I will discuss tools, some of the detective work, and some of the conclusions.

Some of this information is a “well known craft”. Some of it is a “black art”.

The Well Known Craft

Minimizing maximum **interrupts disabled duration** is often a crucial part of ensuring adequate real time latency.

The Well Known Craft

Measuring **interrupts disabled duration**

Can use existing tools

The next slides are an example report of the `ftrace irqsoff` tracer, with the `latency-format` option enabled, from `Documentation/trace/ftrace.txt`

latency: 50 us, #101/101, CPU#0 | (M:preempt VP:0, KP:0, SP:0 HP:0 #P:2)

task: ls-4339 (uid:0 nice:0 policy:0 rt_prio:0)

=> started at: __alloc_pages_internal

=> ended at: __alloc_pages_internal

```
#          _-----=> CPU#
#          /_-----=> irqs-off
#          | /_-----=> need-resched
#          || /_-----=> hardirq/softirq
#          ||| /_-----=> preempt-depth
#          |||| /
#          |||||
#          ||||| delay
#  cmd      pid  ||||| time | caller
#  \      /  ||||| \  | /
ls-4339  0...1  0us+: get_page_from_freelist (__alloc_pages_internal)
ls-4339  0d..1  3us : rmqueue_bulk (get_page_from_freelist)
ls-4339  0d..1  3us : _spin_lock (rmqueue_bulk)
ls-4339  0d..1  4us : add_preempt_count (_spin_lock)
ls-4339  0d..2  4us : __rmqueue (rmqueue_bulk)
ls-4339  0d..2  5us : __rmqueue_smallest (__rmqueue)
ls-4339  0d..2  5us : __mod_zone_page_state (__rmqueue_smallest)
ls-4339  0d..2  6us : __rmqueue (rmqueue_bulk)
ls-4339  0d..2  6us : __rmqueue_smallest (__rmqueue)
ls-4339  0d..2  7us : __mod_zone_page_state (__rmqueue_smallest)
ls-4339  0d..2  7us : __rmqueue (rmqueue_bulk)
ls-4339  0d..2  8us : __rmqueue_smallest (__rmqueue)
[... ]
ls-4339  0d..2  46us : __rmqueue_smallest (__rmqueue)
ls-4339  0d..2  47us : __mod_zone_page_state (__rmqueue_smallest)
```

latency: 50 us, #101/101, CPU#0 | (M:preempt VP:0, KP:0, SP:0 HP:0 #P:2)

```
#          | | | | |      delay
#  cmd      pid  | | | | |  time  |  caller
#  \      /  | | | | |  \    |  /
  ls-4339  0d..1  0us+: get_page_from_freelist (__alloc_pages_internal)
  ls-4339  0d..1  3us : rmqueue_bulk (get_page_from_freelist)
  ls-4339  0d..1  3us : _spin_lock (rmqueue_bulk)
  ls-4339  0d..1  4us : add_preempt_count (_spin_lock)
  ls-4339  0d..2  4us : __rmqueue (rmqueue_bulk)
  ls-4339  0d..2  5us : __rmqueue_smallest (__rmqueue)
  ls-4339  0d..2  5us : __mod_zone_page_state (__rmqueue_smallest)
  ls-4339  0d..2  6us : __rmqueue (rmqueue_bulk)
  ls-4339  0d..2  6us : __rmqueue_smallest (__rmqueue)
  ls-4339  0d..2  7us : __mod_zone_page_state (__rmqueue_smallest)
  ls-4339  0d..2  7us : __rmqueue (rmqueue_bulk)
  ls-4339  0d..2  8us : __rmqueue_smallest (__rmqueue)
[... ]
  ls-4339  0d..2  46us : __rmqueue_smallest (__rmqueue)
  ls-4339  0d..2  47us : __mod_zone_page_state (__rmqueue_smallest)
  ls-4339  0d..2  47us : __rmqueue (rmqueue_bulk)
  ls-4339  0d..2  48us : __rmqueue_smallest (__rmqueue)
  ls-4339  0d..2  48us : __mod_zone_page_state (__rmqueue_smallest)
  ls-4339  0d..2  49us : _spin_unlock (rmqueue_bulk)
  ls-4339  0d..2  49us : sub_preempt_count (_spin_unlock)
  ls-4339  0d..1  50us : get_page_from_freelist (__alloc_pages_internal)
  ls-4339  0d..2  51us : trace_hardirqs_on (__alloc_pages_internal)
```

My Motivation, part 1

Minimizing maximum **interrupts disabled duration** is often a crucial part of ensuring adequate real time latency.

A significant contributor to maximum interrupts disabled duration in many of my projects has been `do_local_timer()`.

Capturing do_local_timer() data

Custom tool (the “lite tracer”)

Similar to the ftrace irqsoff tracer,
but lower overhead and additional features

- Reports each new high-water critical period (eg IRQs disabled duration)
- Reports frequency distribution of duration of instrumented code paths (histogram)

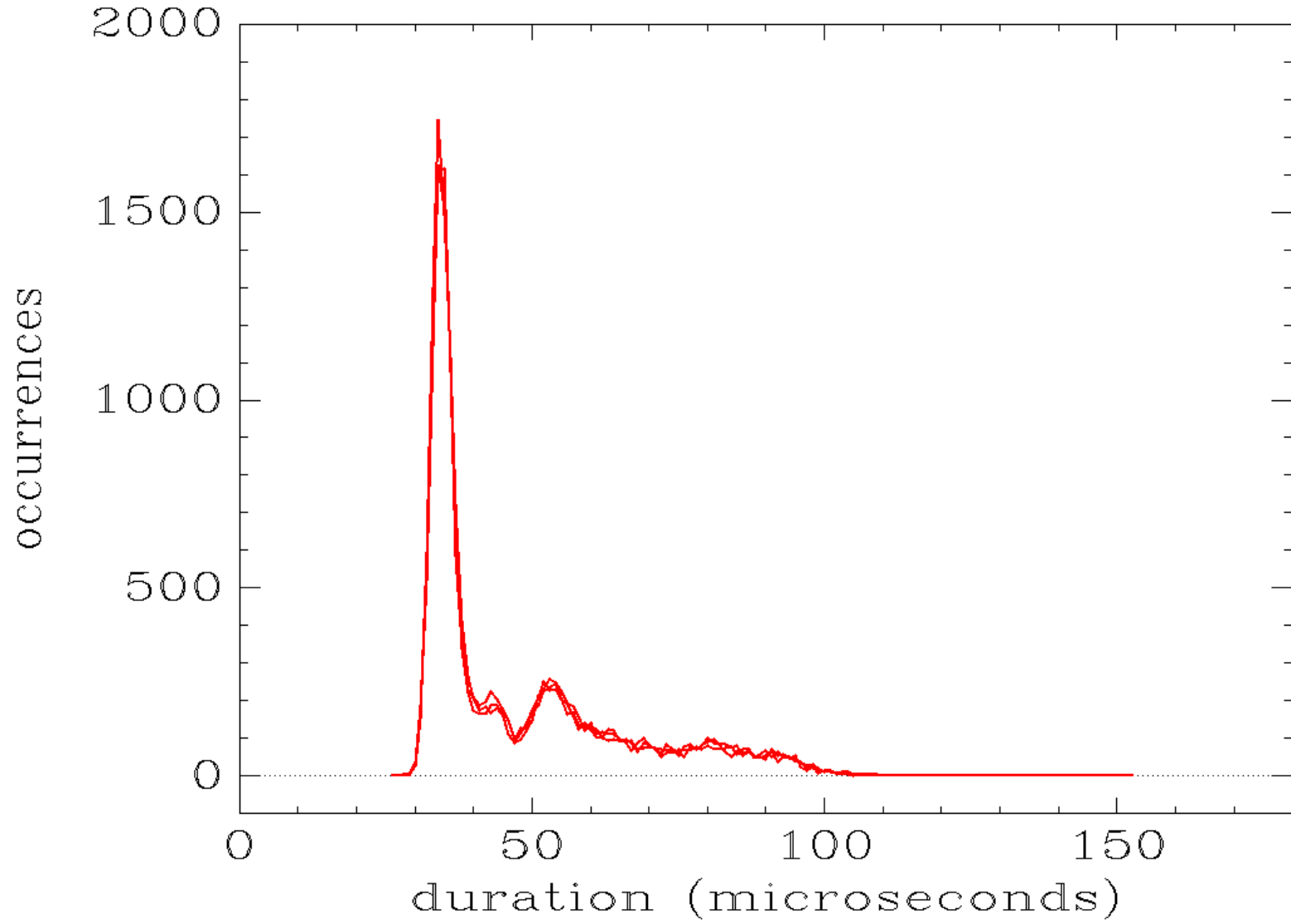
Capturing do_local_timer() data

The following graphs show an example distribution of do_local_timer() durations from a modified 2.6.23.17-rt14.

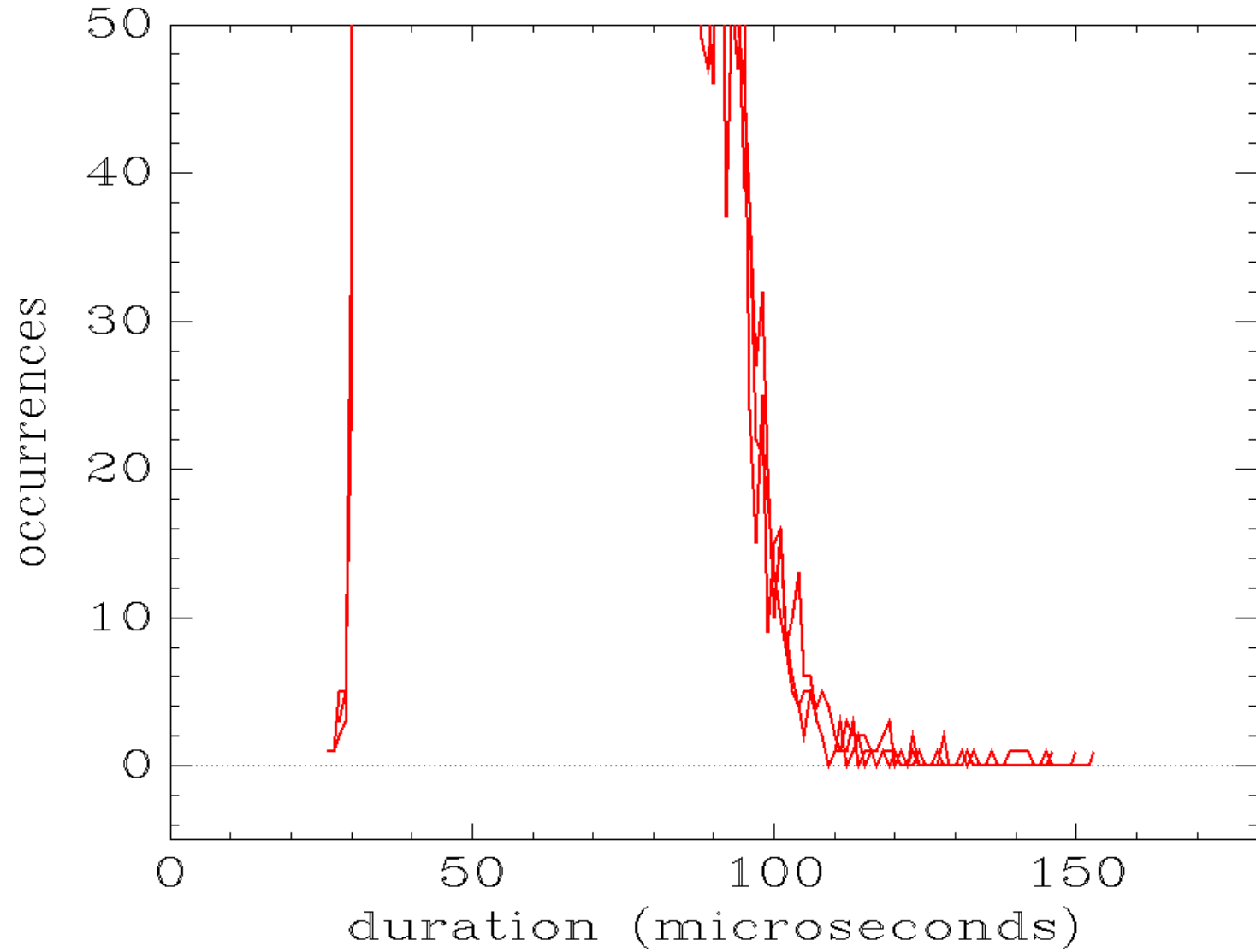
Things to notice about the graphs:

- the consistency between test runs
- the tails -- the elusive 10%
- maximum non-tail value vs. maximum tail value

2.6.23 00066 do_local_timer
cpu 0



2.6.23 00066 do_local_timer
cpu 0



Digression: do_local_timer() info

(on the boards used for this presentation)

- Triggered on each cpu by a per cpu irq source
- Periodic, once every 10 msec (100 times/sec)
- Thus ~10 msec of other activity which can impact the i-cache between each occurrence of do_local_timer()
- Executes in interrupt context with interrupts disabled

Moving Forward

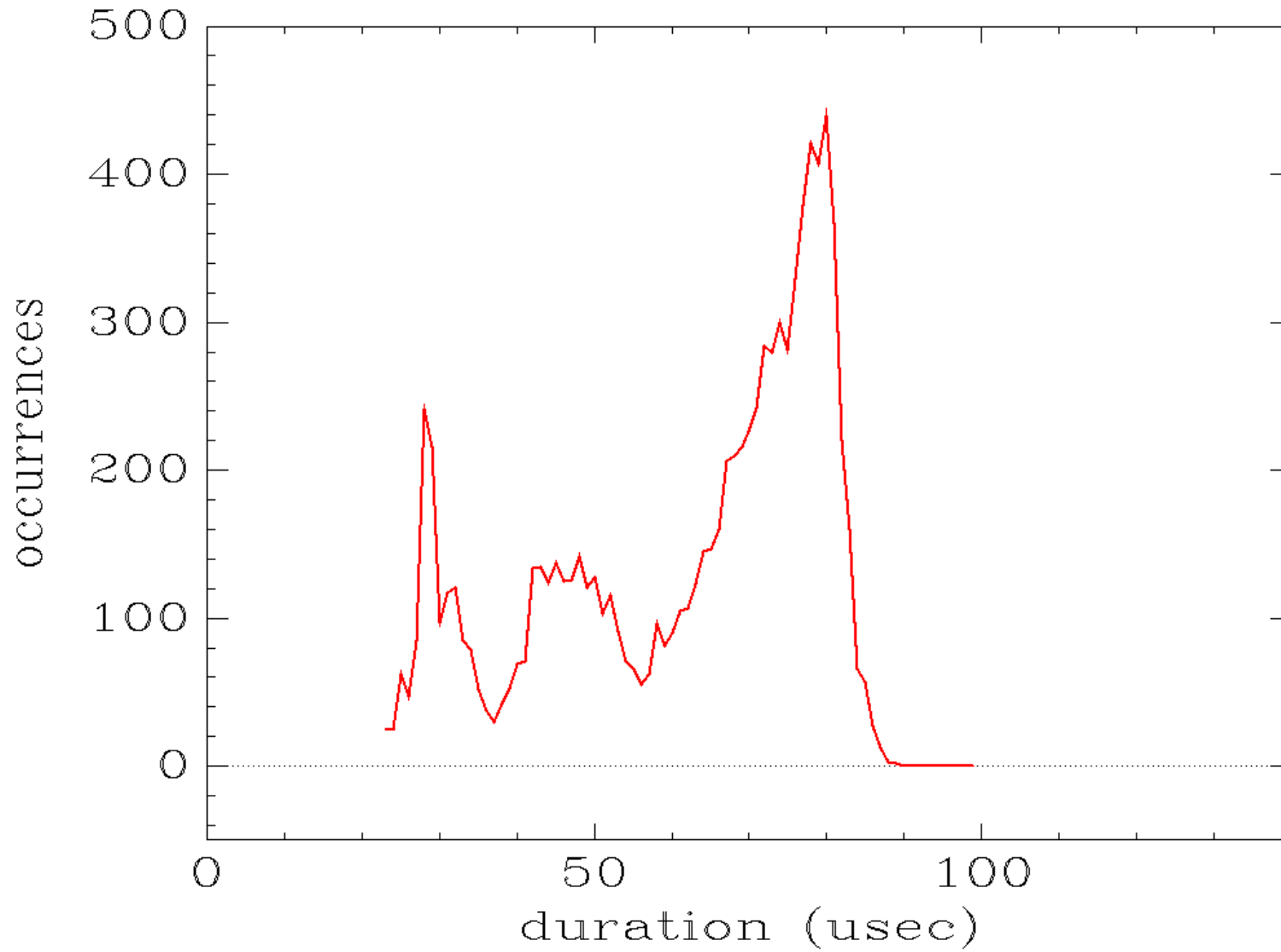
More recent kernel: modified 2.6.29.6-rt14

A different processor on a different board

`do_local_timer()` is still an important portion of maximum interrupts disabled duration

The `do_local_timer()` duration frequency graph looks different, yet still has a tail

2.6.29 27 do_local_timer
cpu 0



My Motivation, part 2

My colleagues added PMU counters to the lite tracer reports of maximum IRQ disabled duration:

```
#mark          : ----- 9  max:          145 -----
# start eip0: c03a8294 __irq_svc+0x34
#      6 do_local_timer+0x40  284718 284102 27556
#     96 do_local_timer+0x80  317730 308527 30419
#     98 _text+0x98          318015 308634 30486
#    138 _text+0xc0          332232 317411 32512
# end   eip0: c03a5528 preempt_schedule_irq+0x48
```


My Motivation, part 2

```
#          6 do_local_timer+0x40  284718 284102 27556
#          96 do_local_timer+0x80 317730 308527 30419
```

```
elapsed usec          96 -          6 =          90
```

Example of PMU counters:

```
cycle count          317730 - 284718 = 33012
I-stall cycles       308527 - 284102 = 24425  74%
Instructions          30419 - 27556 = 2863
```

```
          cycles per instruction  33012 / 2863 = 11.5
I-stall  cycles per instruction  24425 / 2863 = 8.5
```

My First PMU Data

Collecting `do_local_timer()` PMU data for some instances of maximum interrupts disabled provided the following data and graph.

(All functions other than `do_local_timer()` are edited out of the following report.)

```

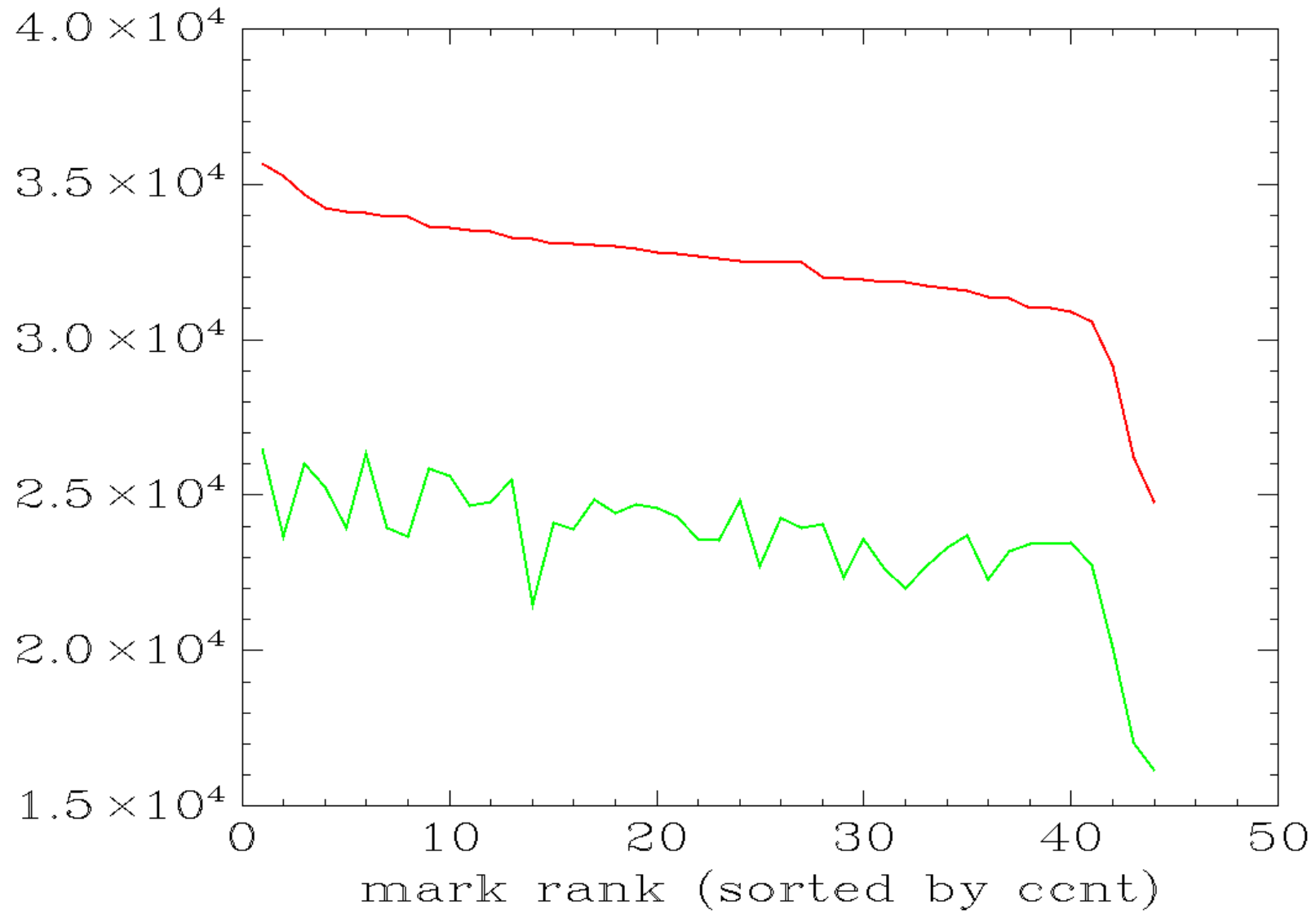
#mark          : ----- 1 max:          4 -----
#mark          : ----- 2 max:          7 -----
#mark          : ----- 3 max:          8 -----
#mark          : ----- 4 max:         30 -----
#mark          : ----- 5 max:         97 -----
#  6 c002830c do_local_timer+0x40 4080208 1572163 1015603
# 93 c002834c do_local_timer+0x80 4112153 1594506 1018468
#mark          : ----- 6 max:         126 -----
#  6 c002830c do_local_timer+0x40 8063703 3197391 1943556
# 93 c002834c do_local_timer+0x80 8095345 3220720 1946403
#mark          : ----- 7 max:         132 -----
#  6 c002830c do_local_timer+0x40 71787138 11313175 5711988
# 92 c002834c do_local_timer+0x80 71818448 11336389 5714851
#mark          : ----- 8 max:         138 -----
#  6 c002830c do_local_timer+0x40 75760529 13366792 6479322
# 93 c002834c do_local_timer+0x80 75792438 13390386 6482185
#mark          : ----- 9 max:         145 -----
#  6 c002830c do_local_timer+0x40 382284718 112284102 51627556
# 96 c002834c do_local_timer+0x80 382317730 112308527 51630419

```

usec	ccnt	i-stall cycles
-----	-----	-----
96	35641	26480
95	35242	23675
93	34644	26020
94	34242	25255
94	34107	23934
...		
85	31006	23436
85	30911	23472
83	30576	22745
81	29142	20106
72	26209	17026
68	24752	16127

irqs_off_27

red: cnt green: I-cache stall cycles



Observations

I-cache misses and I-cache stall cycles for the `do_local_timer()` code path are **large** when `do_local_timer()` duration is **large**.

I-cache misses and I-cache stall cycles for the `do_local_timer()` code path are **smaller** when `do_local_timer()` duration is **smaller**.

Observations, Question

I-cache misses and I-cache stall cycles for the `do_local_timer()` code path are **large** when `do_local_timer()` duration is **large**.

I-cache misses and I-cache stall cycles for the `do_local_timer()` code path are **smaller** when `do_local_timer()` duration is **smaller**.

Are I-cache misses the cause of long `do_local_timer()` duration?

More Instrumentation Needed

Reports of PMU data each time a new maximum interrupts disabled duration occurs provides information about a very small fraction of all occurrences of `do_local_timer()`.

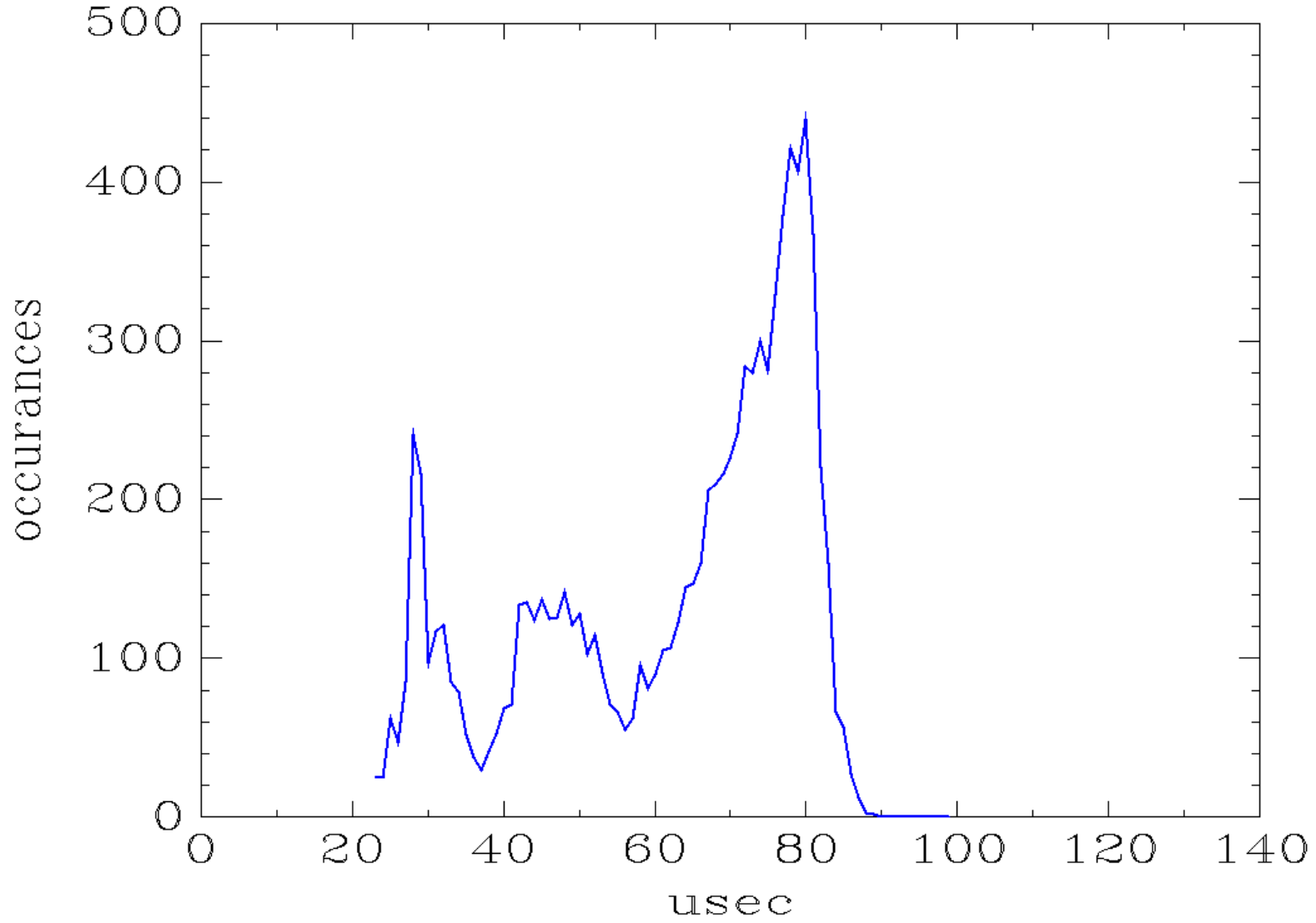
More Instrumentation Needed

Reports of PMU data each time a new maximum interrupts disabled duration occurs provides information about a very small fraction of all occurrences of `do_local_timer()`.

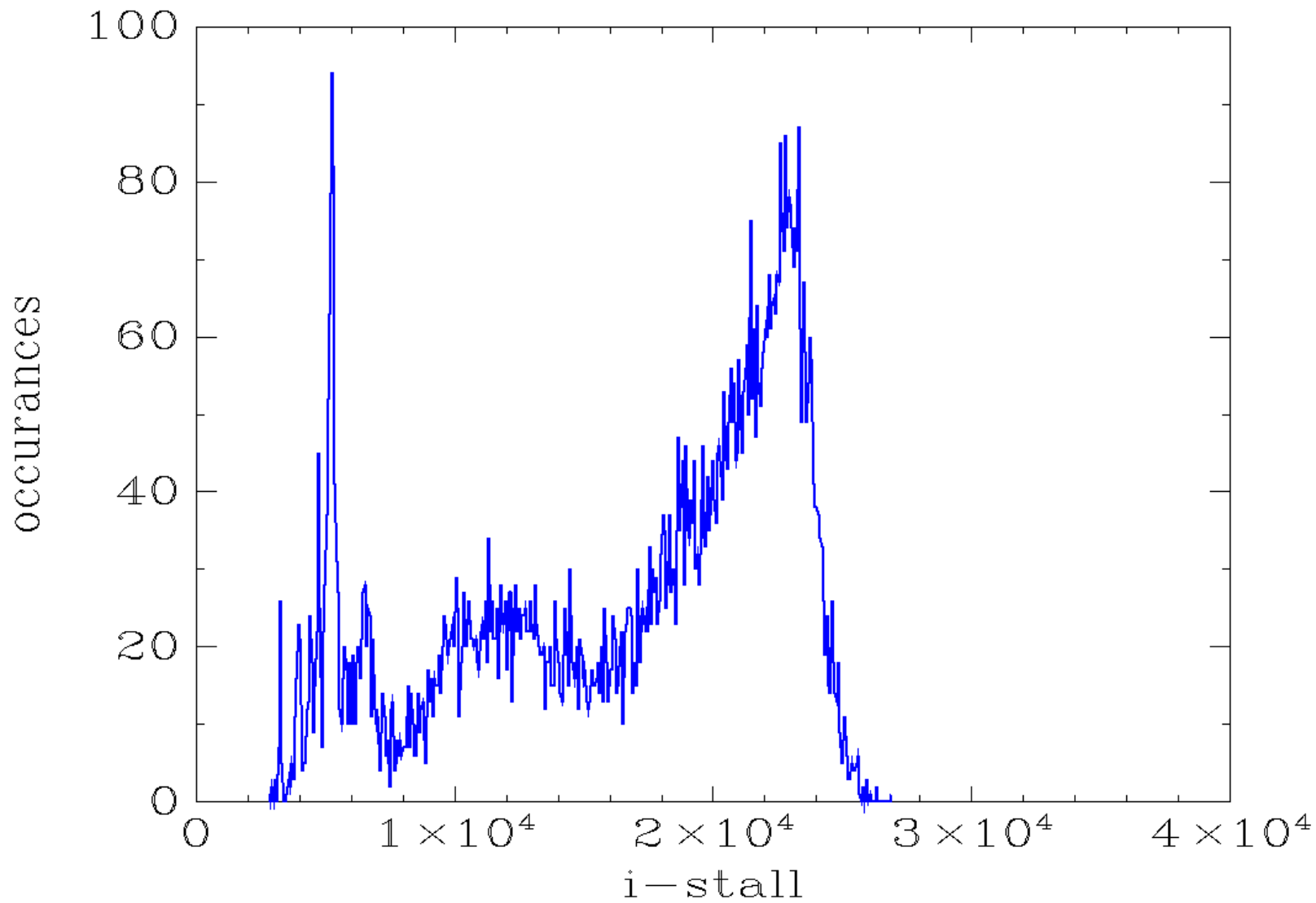
Solution:

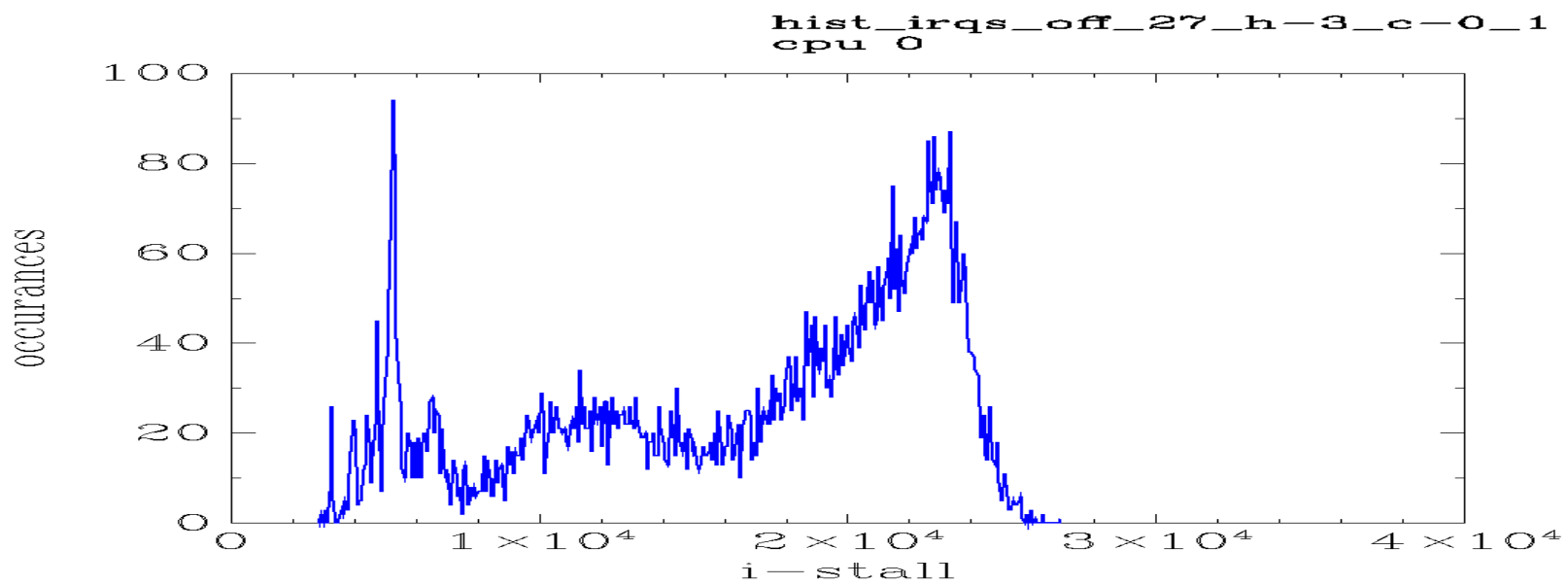
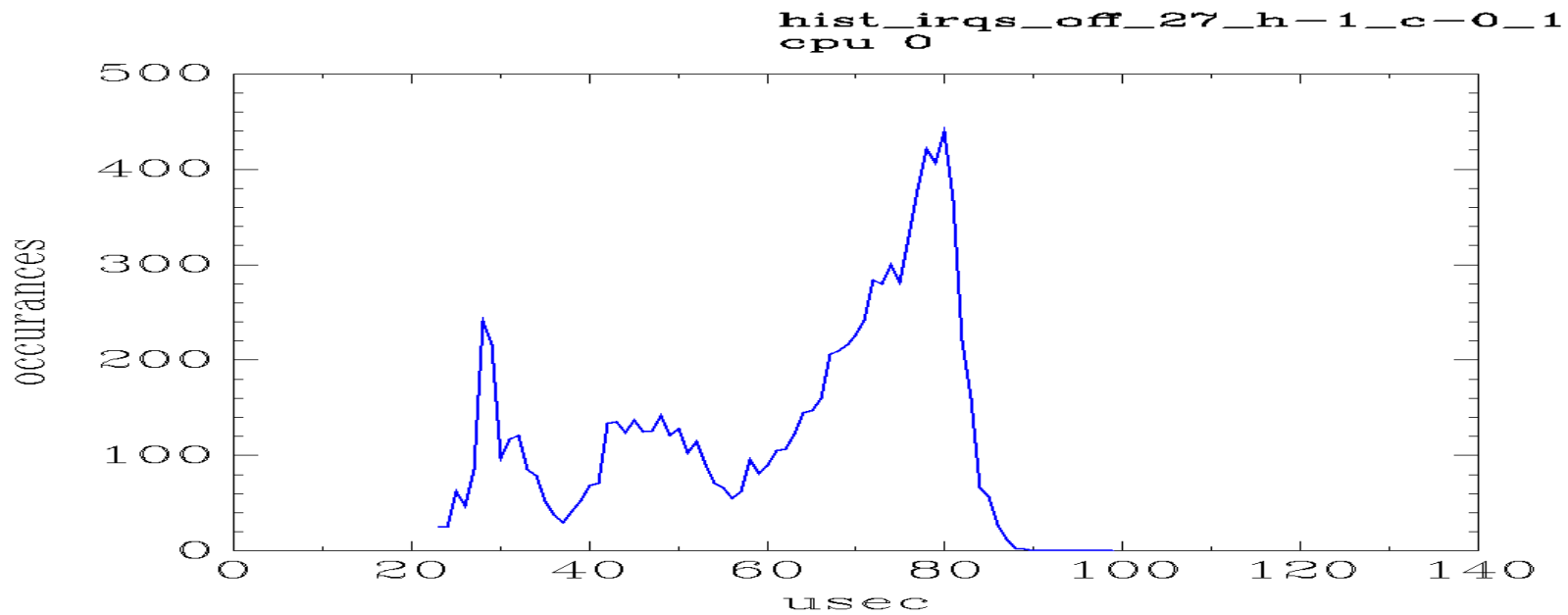
Modify the lite tracer to collect PMU data for each occurrence of `do_local_timer()`.

hist_irqs_off_27_h-1_c-0_1
cpu 0



hist_irqs_off_27_h-3_c-0_1
cpu 0





Conclusions

The graphs of duration (usec) and I-cache stall cycles appear similar.

Conclusions, Theories

The graphs of duration (usec) and I-cache stall cycles appear similar.

Increased `do_local_timer()` duration may be **caused by increased number of instructions.**

Increased `do_local_timer()` duration may be **caused by increased I-cache stall cycles.**

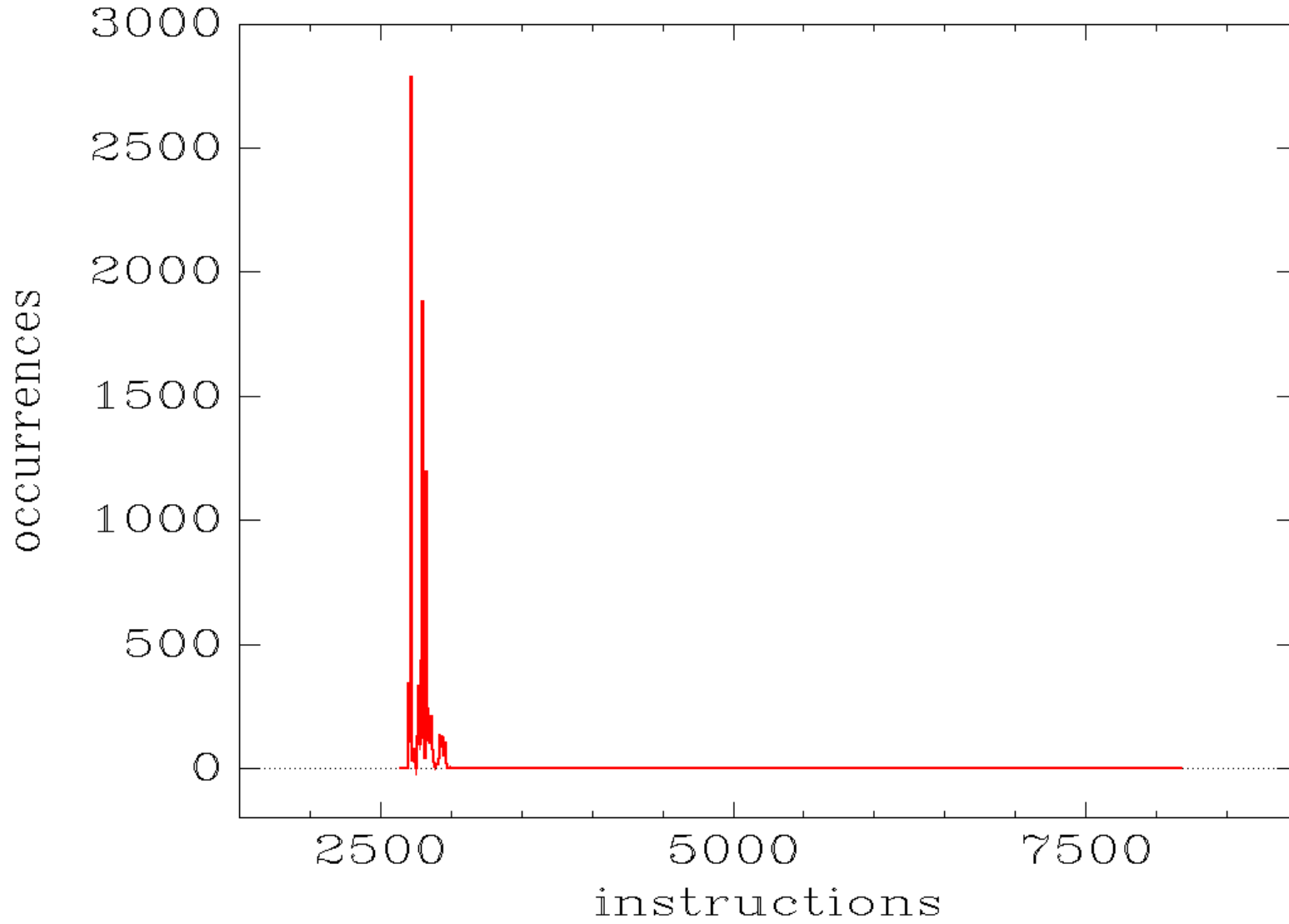
Theories – How To Test?

The graphs of duration (usec) and I-cache stall cycles appear similar.

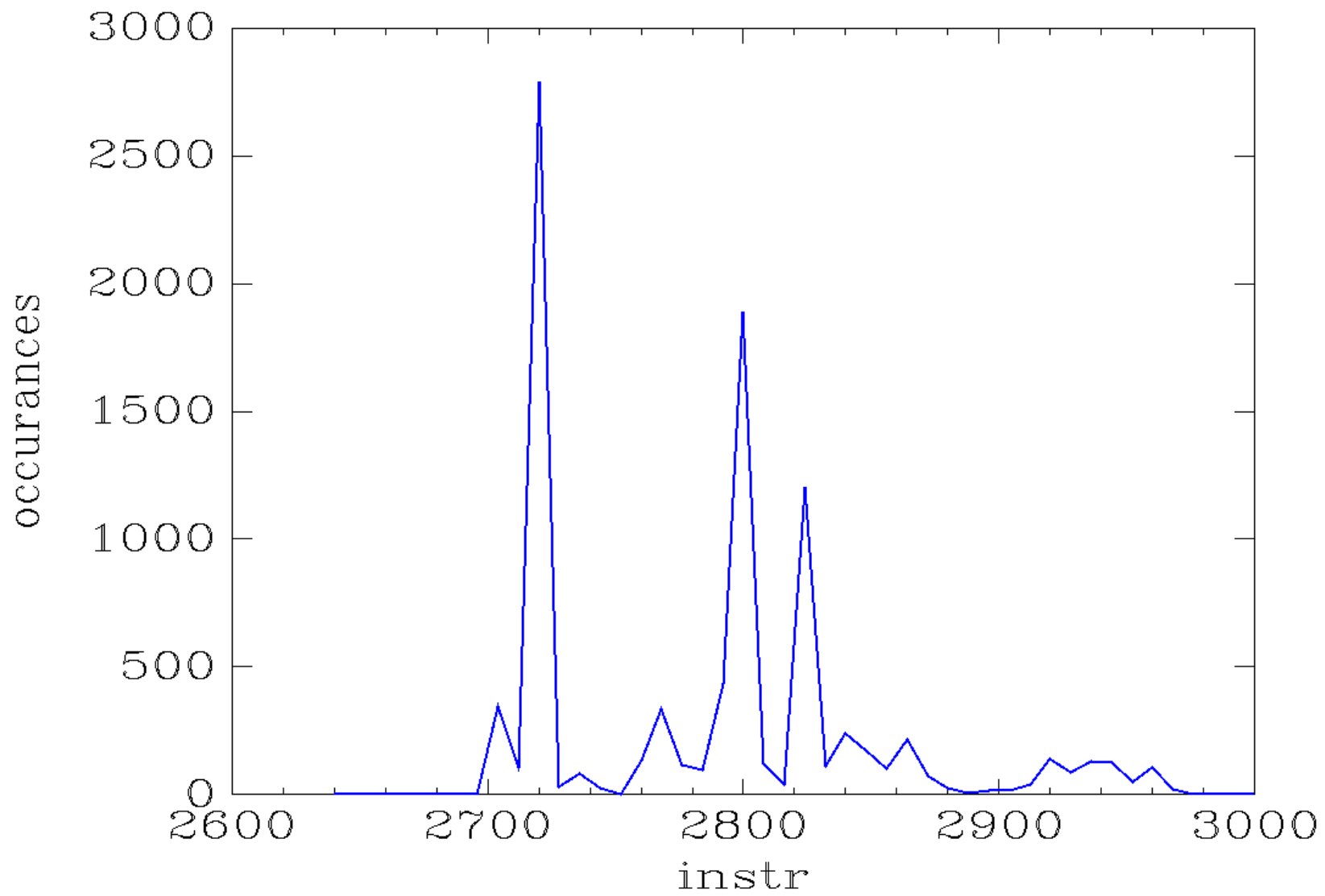
Increased `do_local_timer()` duration may be **caused by increased number of instructions.**

Measure number of instructions executed.

2.6.29 27 do_local_timer
cpu 0



hist_irqs_off_27_h-4_c-0_1
cpu 0



Theories – How To Test?

Increased `do_local_timer()` duration may be **caused by increased number of instructions.**

Measure number of instructions executed.

For 99.7% of the events, the number of instructions executed was in the range 2700 .. 3000.

The large range of `do_local_timer()` duration is not directly explained by number of instructions.

Theories – How To Test?

Increased `do_local_timer()` duration may be **caused by increased I-cache miss stall cycles.**

Invalidate the I-cache before calling `do_local_timer()`. Do I-cache stall cycles increase?

Theories – How To Test?

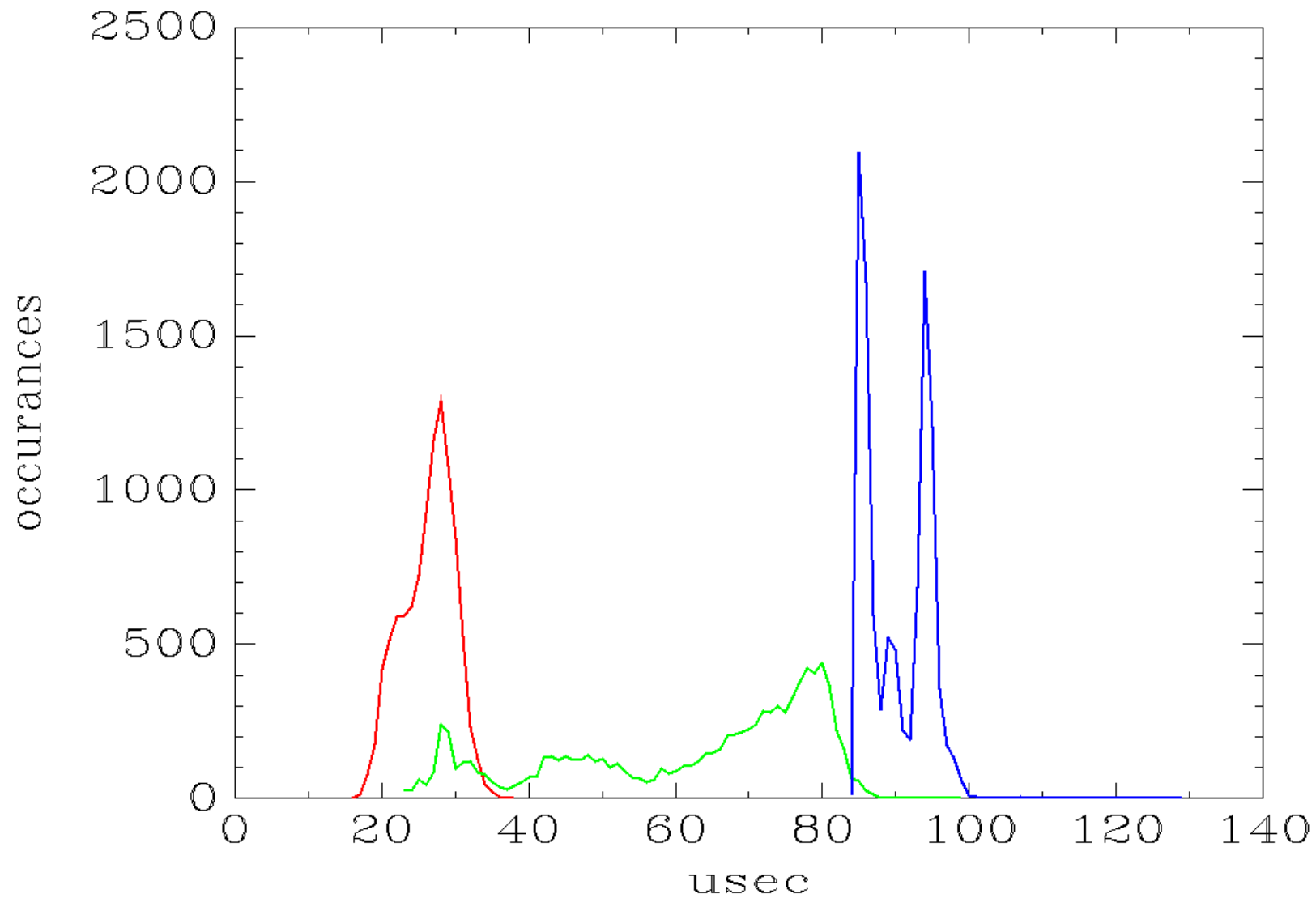
Increased `do_local_timer()` duration may be **caused by increased I-cache miss stall cycles.**

Invalidate the I-cache before calling `do_local_timer()`. Do I-cache stall cycles increase?

Decrease I-cache stall cycles by “pre-fetching” `do_local_timer()` text while interrupts are enabled. Do I-cache stall cycles decrease?

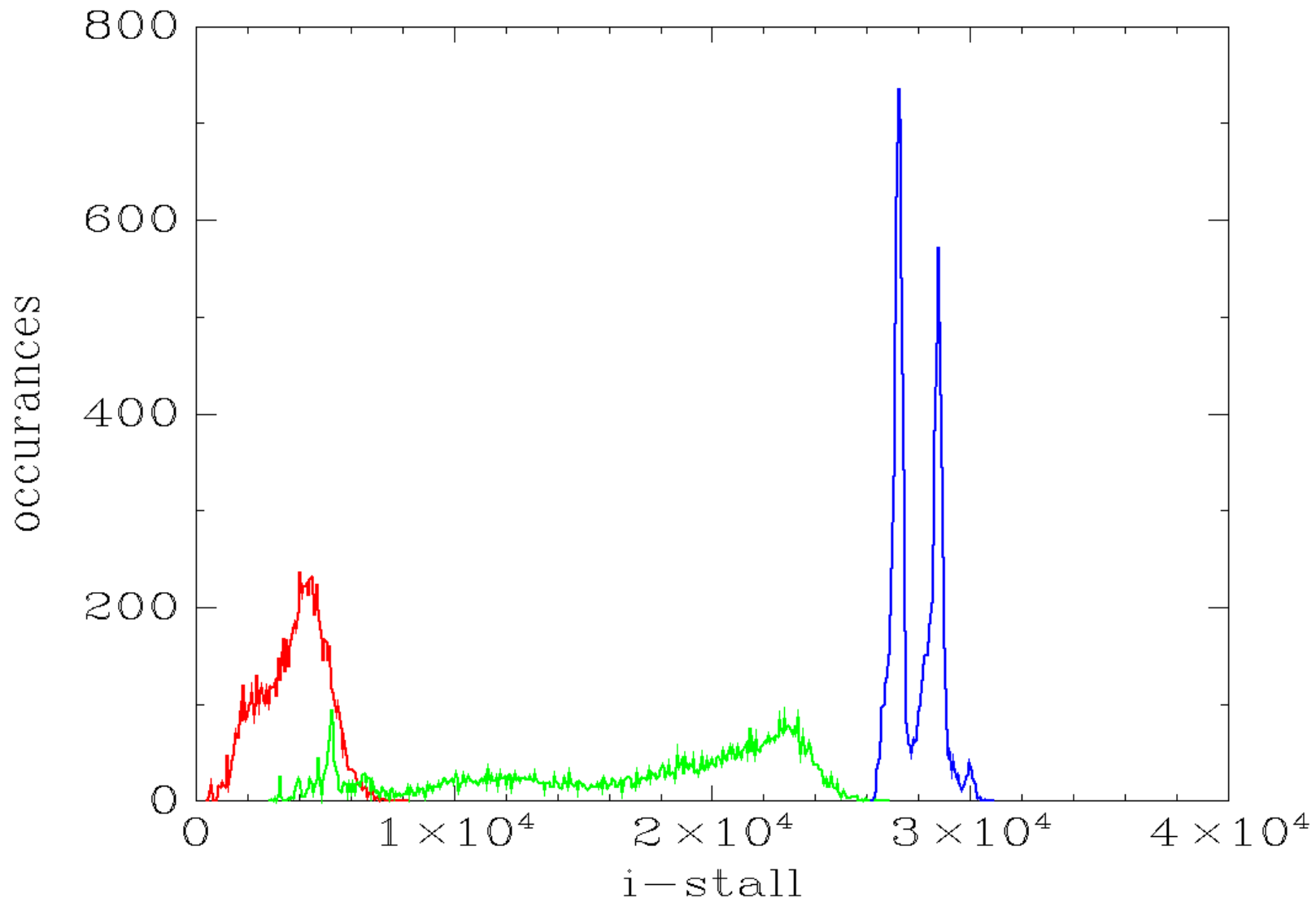
hist_irqs_off_{40-27-41}_h-1_c-0_1

red: prefetch green: normal blue: invalidate I-cache

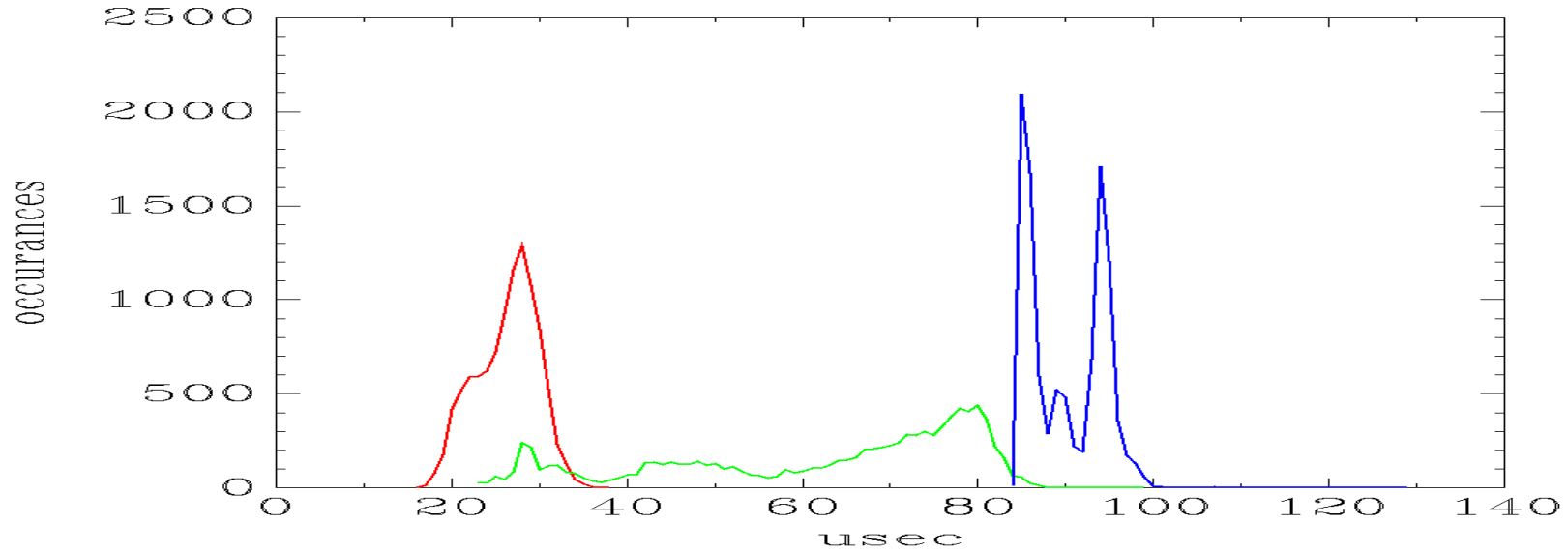


hist_irqs_off_{40-27-41}_h-3_c-0_1

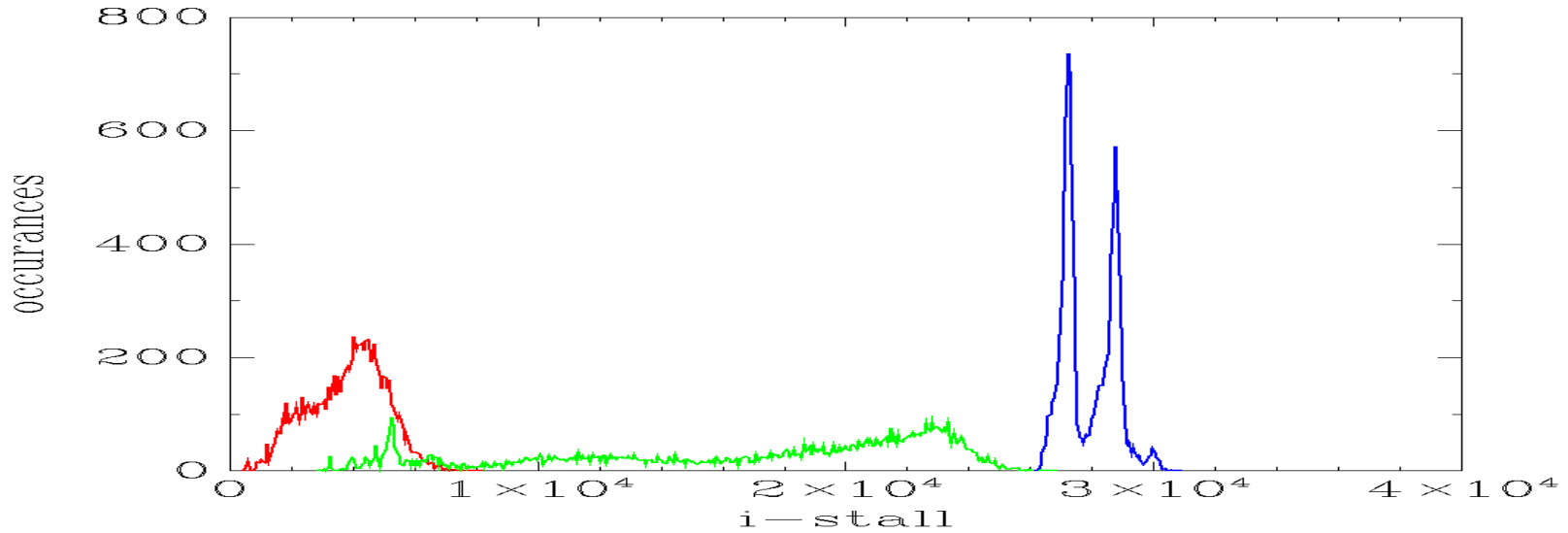
red: prefetch green: normal blue: invalidate I-cache



hist_irqs_off_{40-27-41}_h-1_c-0_1
red: prefetch green: normal blue: invalidate I-cache



hist_irqs_off_{40-27-41}_h-3_c-0_1
red: prefetch green: normal blue: invalidate I-cache



What about IRQs disabled?

Duration of `do_local_timer()` appears to be correlated to I-cache miss stall cycles.

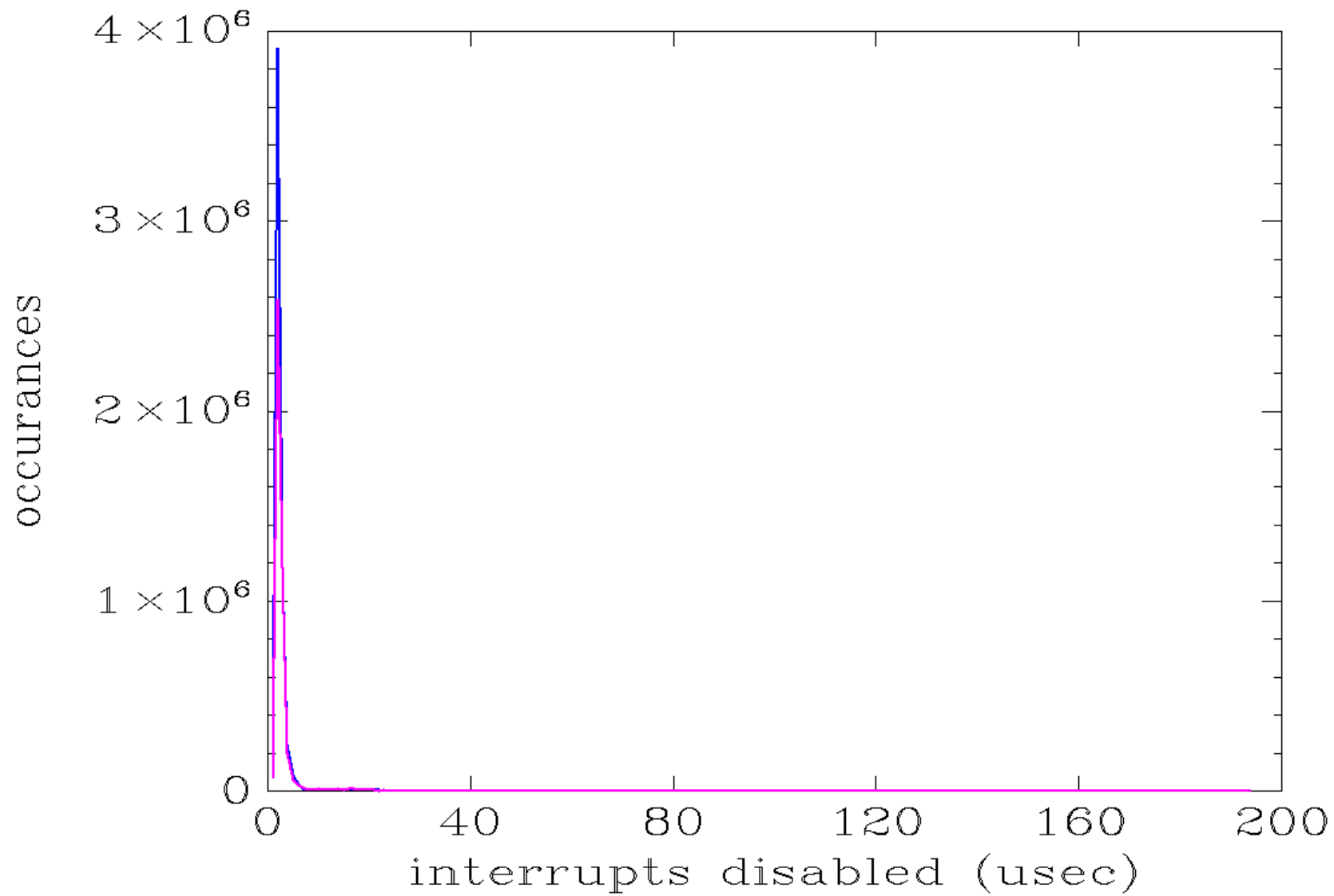
What about IRQs disabled?

Duration of `do_local_timer()` appears to be correlated to I-cache miss stall cycles.

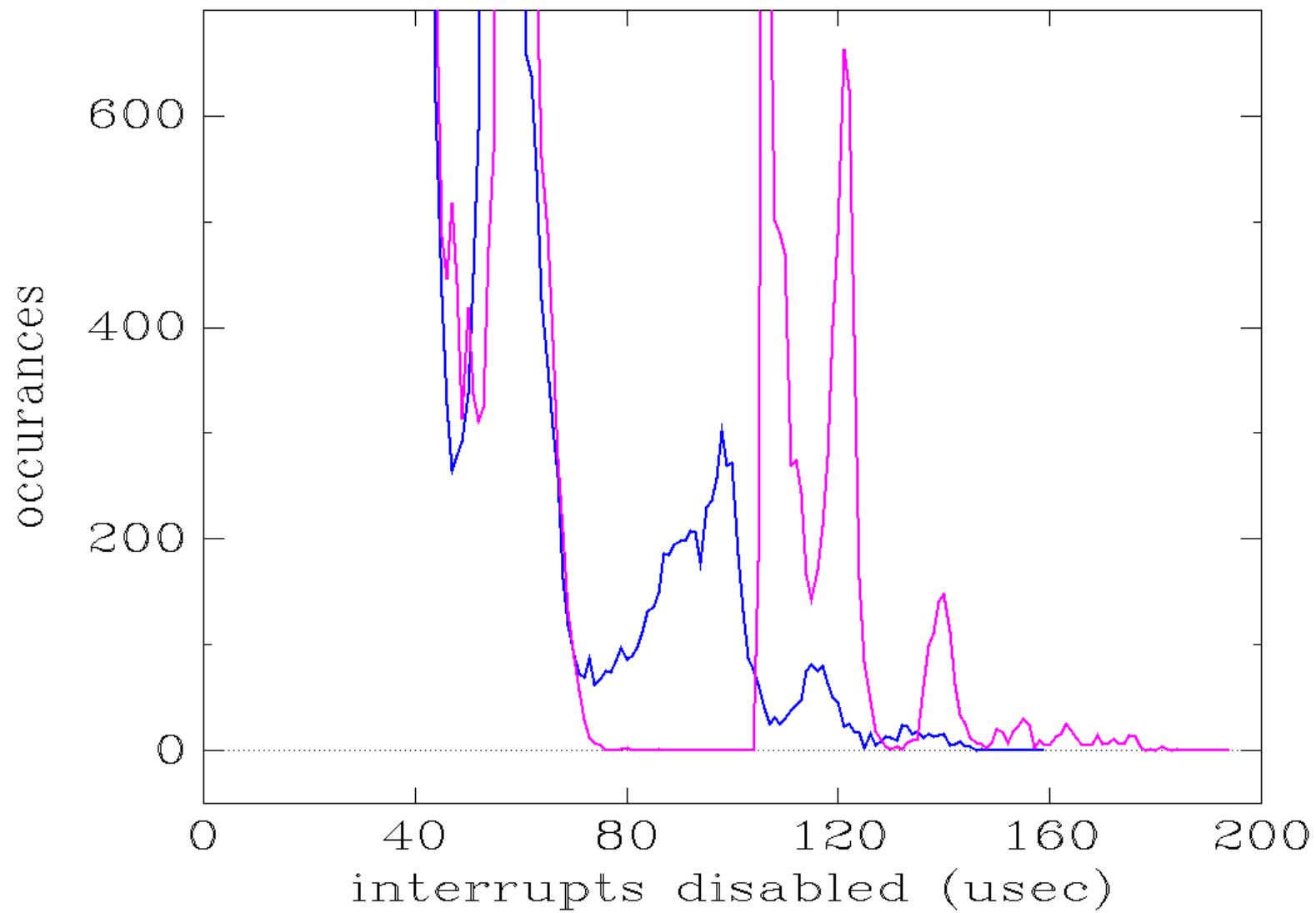
Does this theory help solve the original problem of large maximum interrupt disabled duration?

(Only have data for baseline and worst case. “Pre-fetch” case was not pursued because the processor being tested does not have an instruction to pre-load cache.)

hist_irqs_off_{27,41}_h-0_c-0_1
blue: baseline, magenta: worst case



hist_irqs_off_{27,41}_h-0_c-0_1
blue: baseline, magenta: worst case



Theories – How To Test?

Increased `do_local_timer()` duration may be **caused by increased I-cache miss stall cycles.**

Previous results are consistent with the conclusions, but not proof.

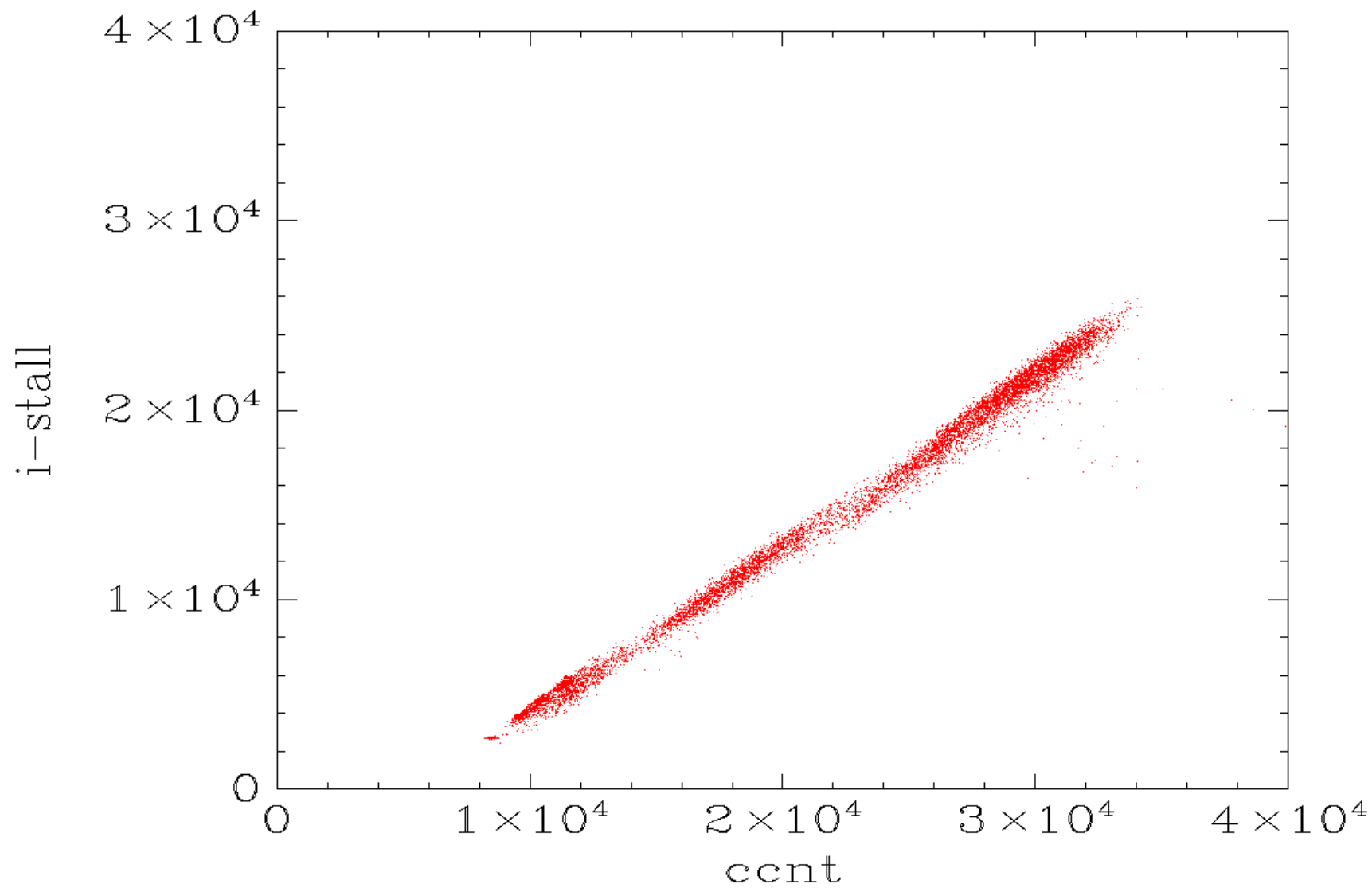
Need more new instrumentation...

Add a trace of PMU data for each event to the lite tracer.

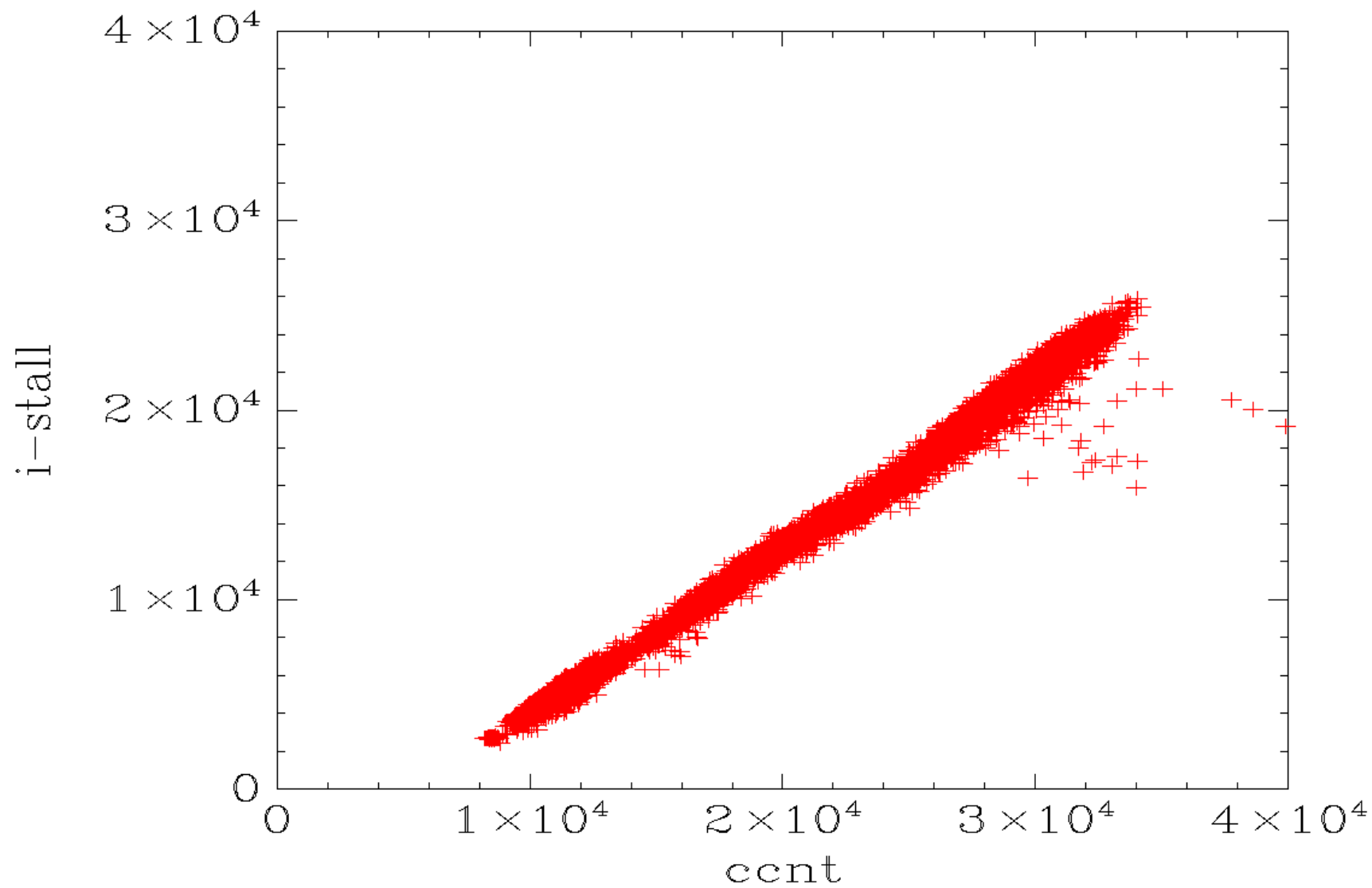
Results

The next graphs are scattergrams of cycle count vs I-cache miss stall cycles

trace_irqs_off_108_f-24_c-0_1_S-0
cpu 0



trace_irqs_off_108_f-24_c-0_1_S-1
cpu 0



Theories – How To Test?

Increased `do_local_timer()` duration may be **caused by increased I-cache miss stall cycles.**

With this data, I was finally felt there was proof of the theory.

Theories – How To Test?

Increased `do_local_timer()` duration may be **caused by increased I-cache miss stall cycles.**

Further instrumentation and analysis of the PMU trace data results in a more nuanced and complex understanding of the `do_local_timer()` duration graphs, with implications for further improvements.

Unfortunately time constraints preclude including that analysis in this presentation.

How To Fix?

Increased `do_local_timer()` duration may be **caused by increased I-cache miss stall cycles.**

The focus of this presentation is finding the causes of the performance problem, not fixing them. But a few possible solutions will be listed.

Then an actual solution will be presented that will lead to more questions about the data.

How To Fix (1)?

Increased `do_local_timer()` duration may be **caused by increased I-cache miss stall cycles.**

Reduce **number of I-cache misses** by increasing I-cache size.

(Exercise for the student:
why are L1 caches typically so small?)

How To Fix (2)?

Increased `do_local_timer()` duration may be **caused by increased I-cache miss stall cycles.**

Reduce **number of I-cache misses** by locking critical period code into I-cache.

Not tested because the processor on the target board does not implement cache locking.

How To Fix (3)?

Increased `do_local_timer()` duration may be **caused by increased I-cache miss stall cycles.**

Decrease **cost of I-cache miss** by placing the critical section code in high speed memory.

Not tested because the processor on the target board does not implement high speed memory.

How To Fix (4)?

Increased `do_local_timer()` duration may be **caused by increased I-cache miss stall cycles.**

Decrease **cost of I-cache miss** by adding an L2 cache.

Move to different hardware

The previous tests were conducted on a processor that does not have an L2 cache.

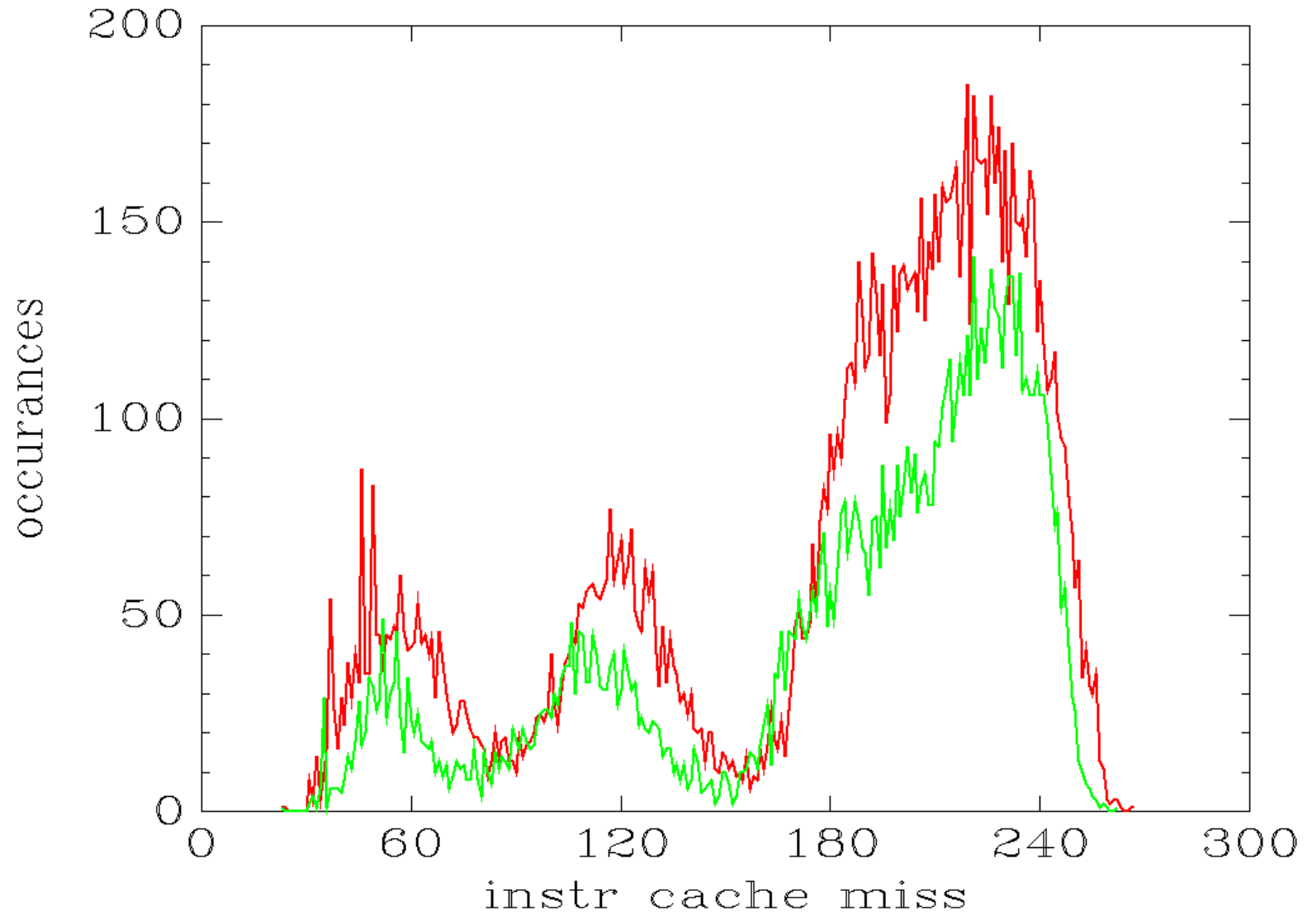
Will try some experiments on a processor with an L2 cache.

The previous graphs are for a 400 Mhz processor.

The following graphs are for a 210 Mhz processor.

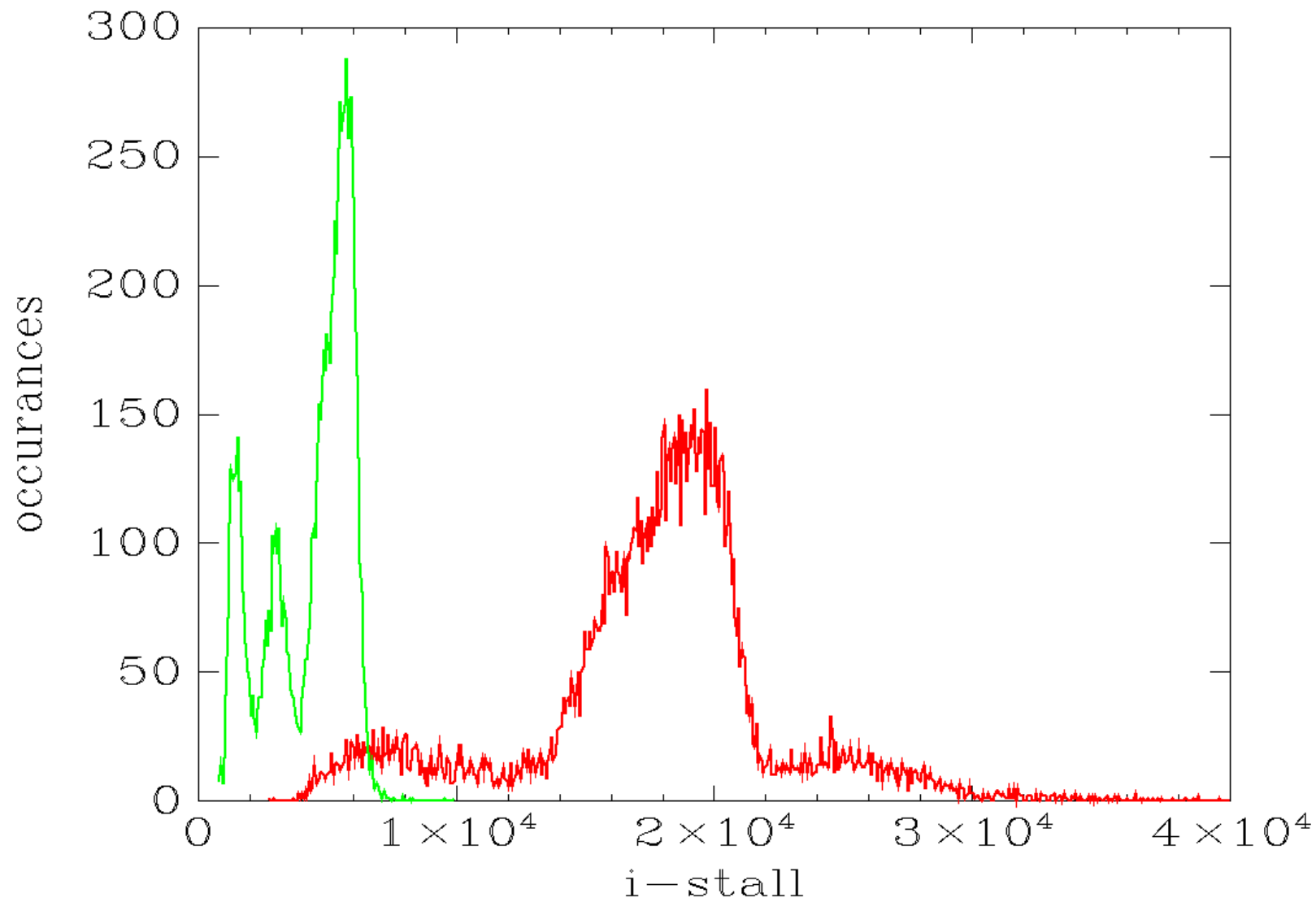
hist_irqs_off_{48-47}_h-3_c-0_1

red: Realview, L2 off green: Realview, L2 on



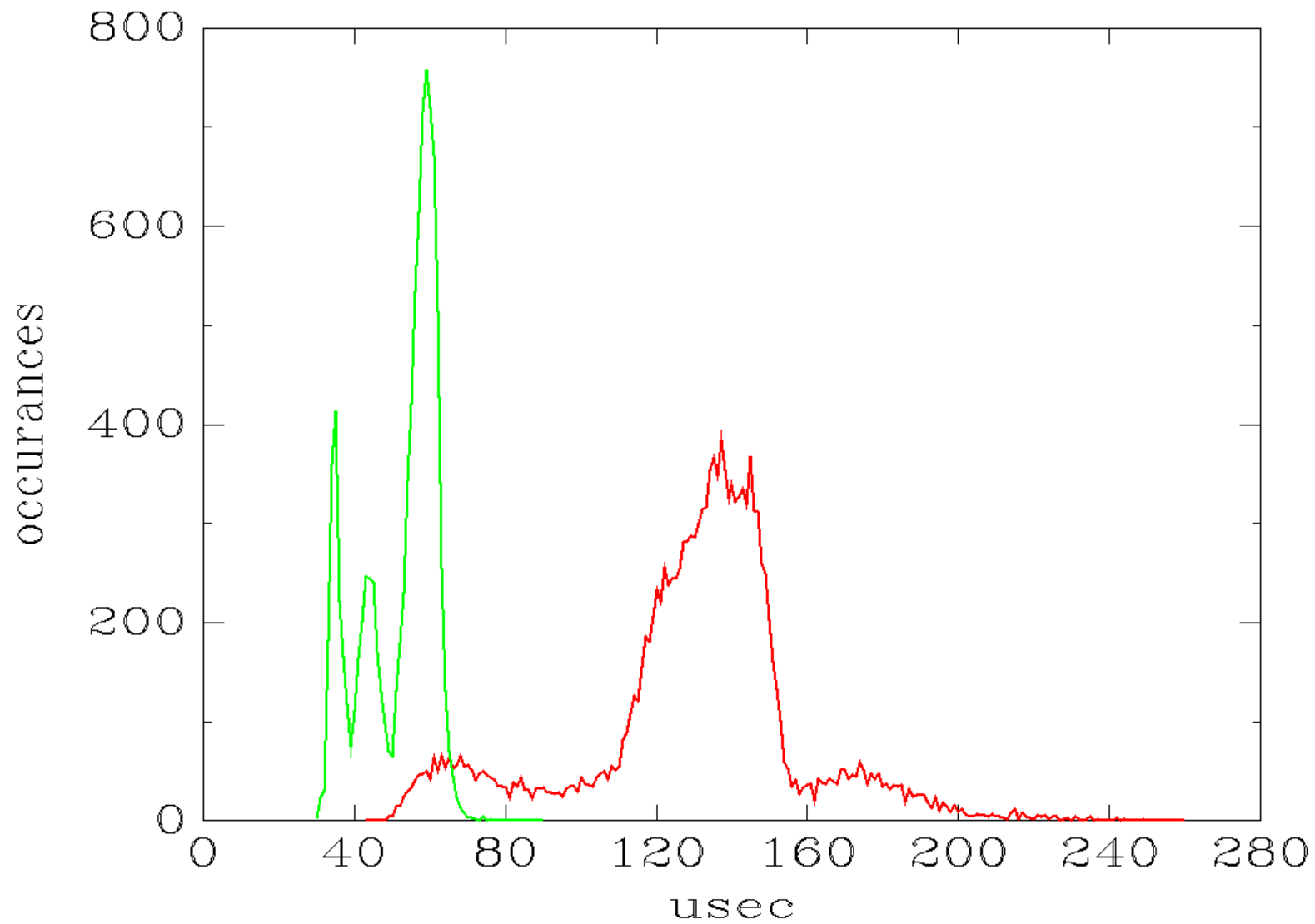
hist_irqs_off_{46-45}_h-3_c-0_1

red: Realview, L2 off green: Realview, L2 on

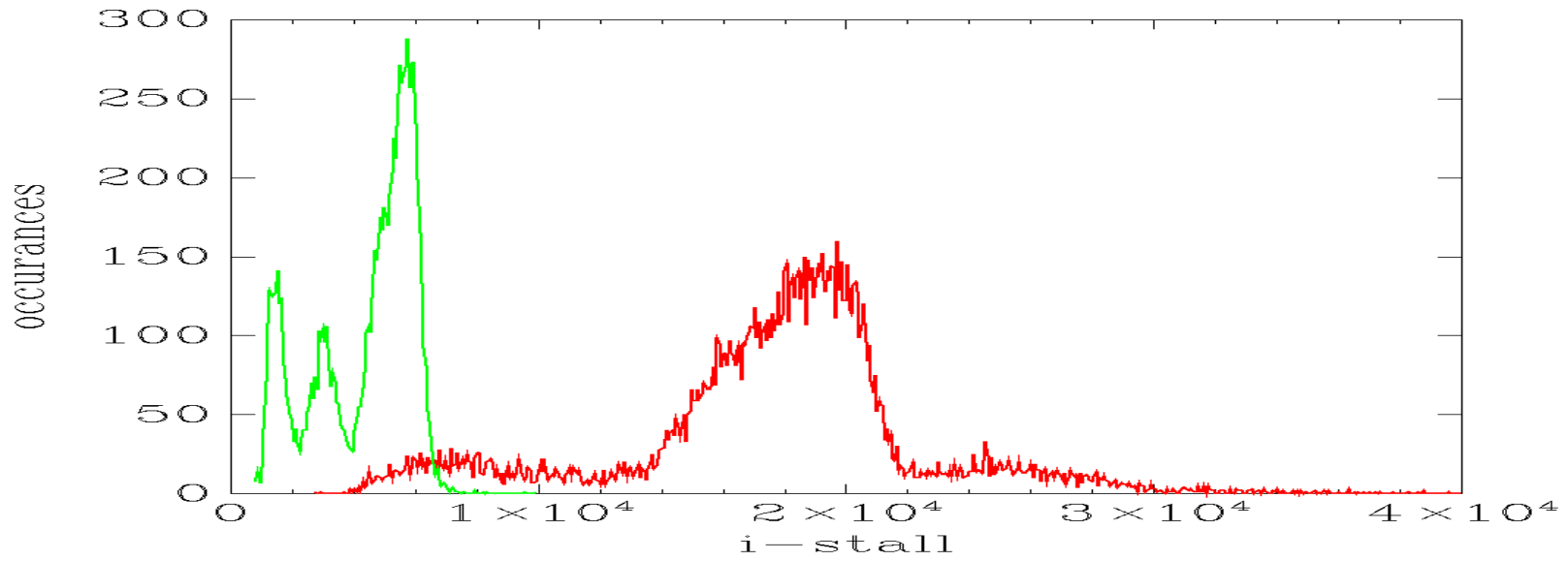


hist_irqs_off_{46-45}_h-1_c-0_1

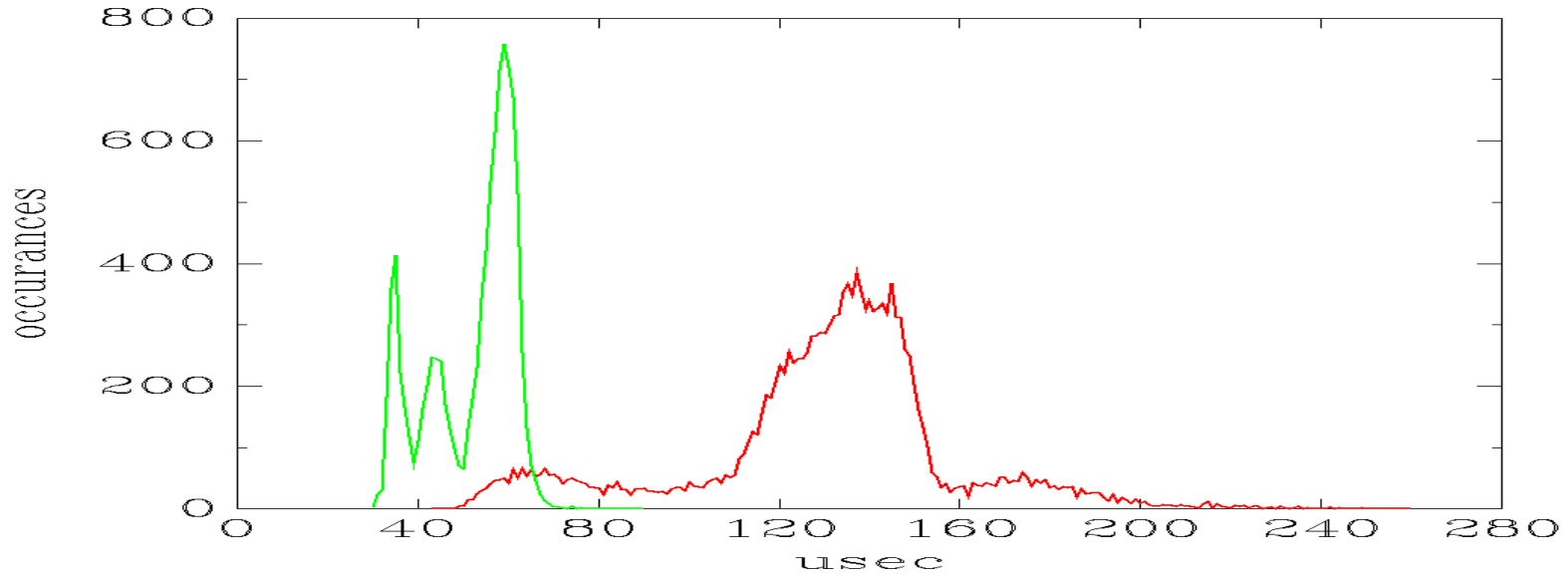
red: Realview, L2 off green: Realview, L2 on



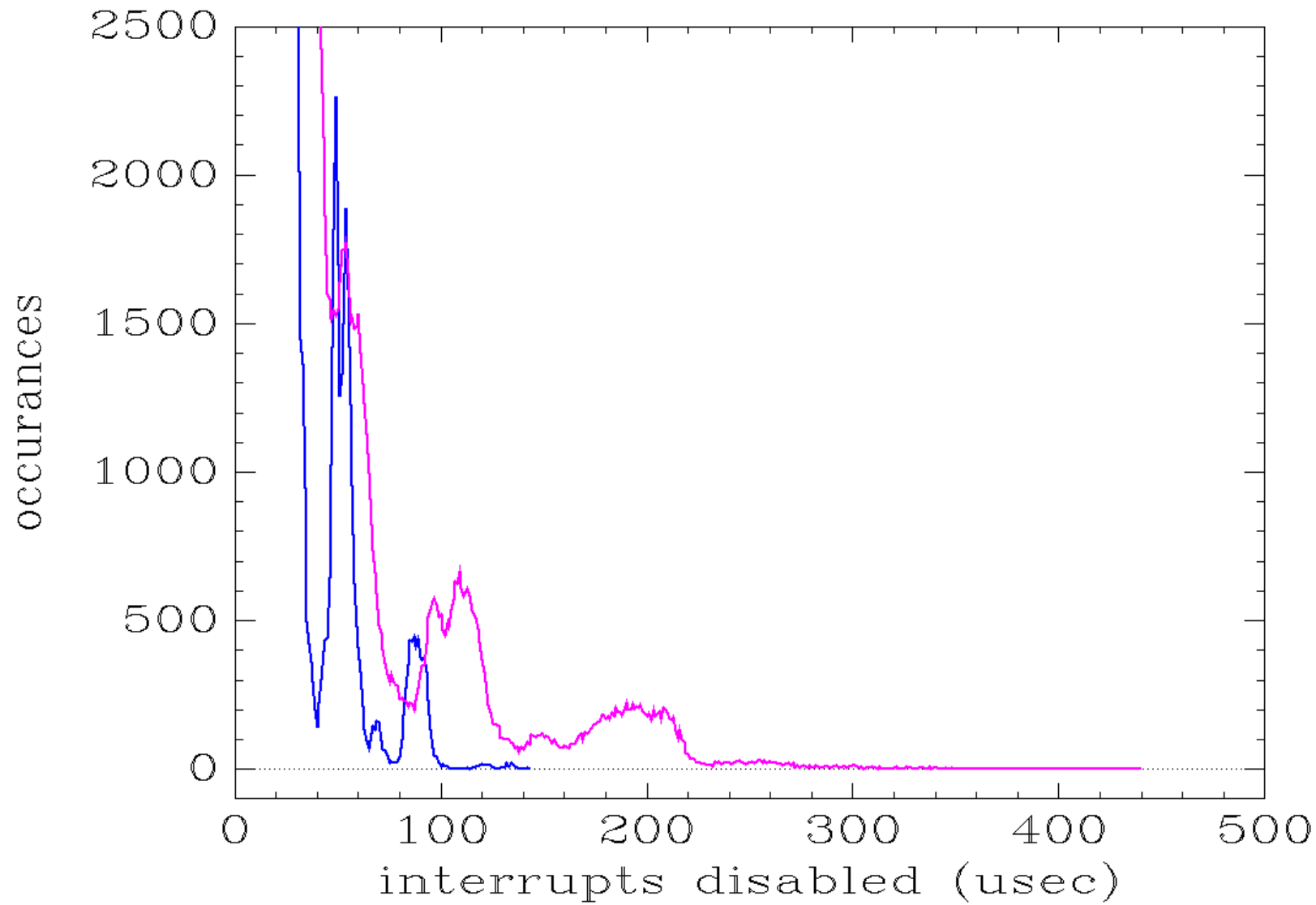
hist_irqs_off_{46-45}_h-3_c-0_1
red: Realview, L2 off green: Realview, L2 on



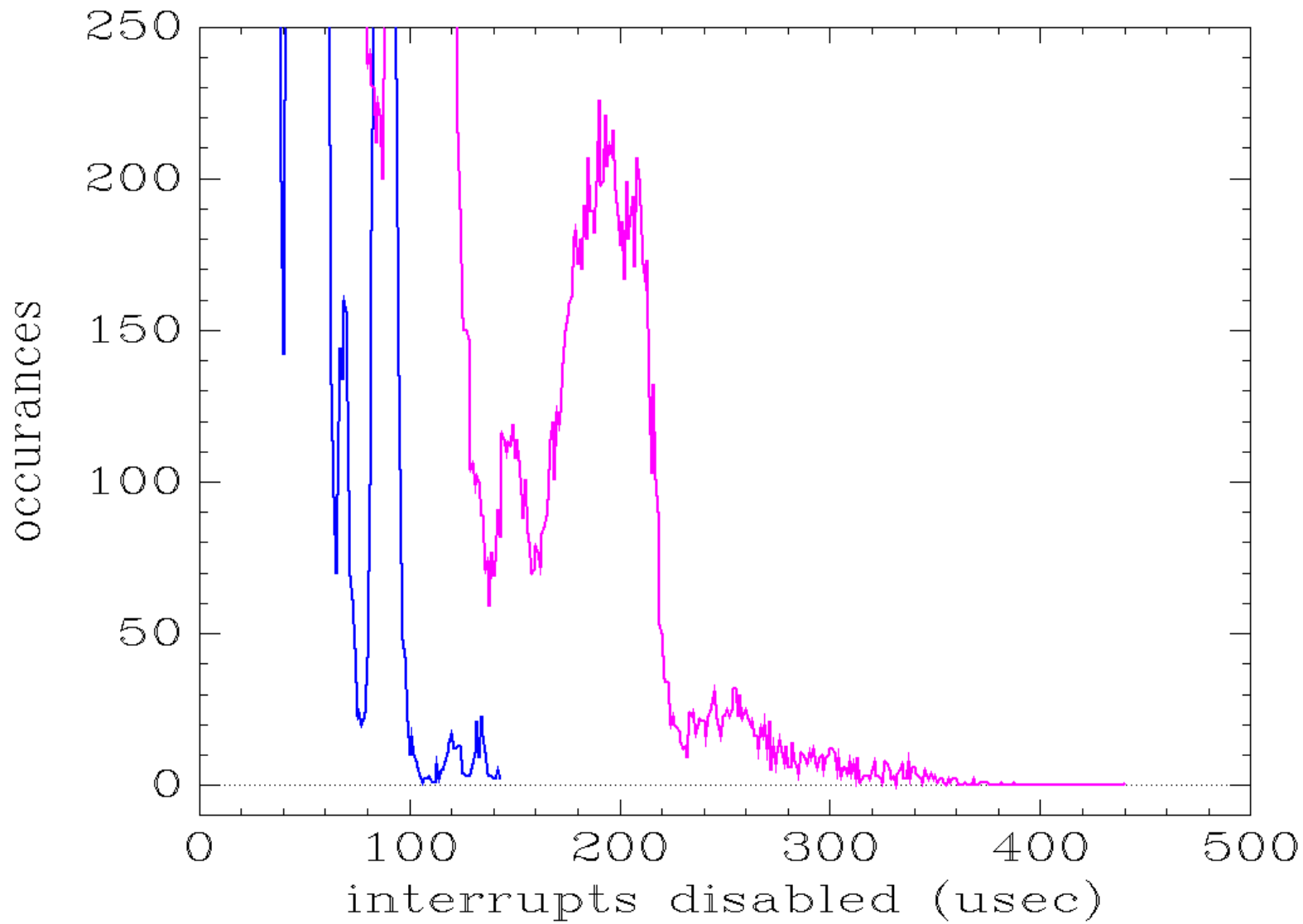
hist_irqs_off_{46-45}_h-1_c-0_1
red: Realview, L2 off green: Realview, L2 on



hist_irqs_off_{46,45}_h-0_c-0_1
blue: L2 enabled, magenta: L2 disabled



hist_irqs_off_{46,45}_h-0_c-0_1
blue: L2 enabled, magenta: L2 disabled



Result

L2 cache is effective for this scenario.

The L1 I-cache misses are often serviced from the L2 cache, which has a much lower latency than memory.

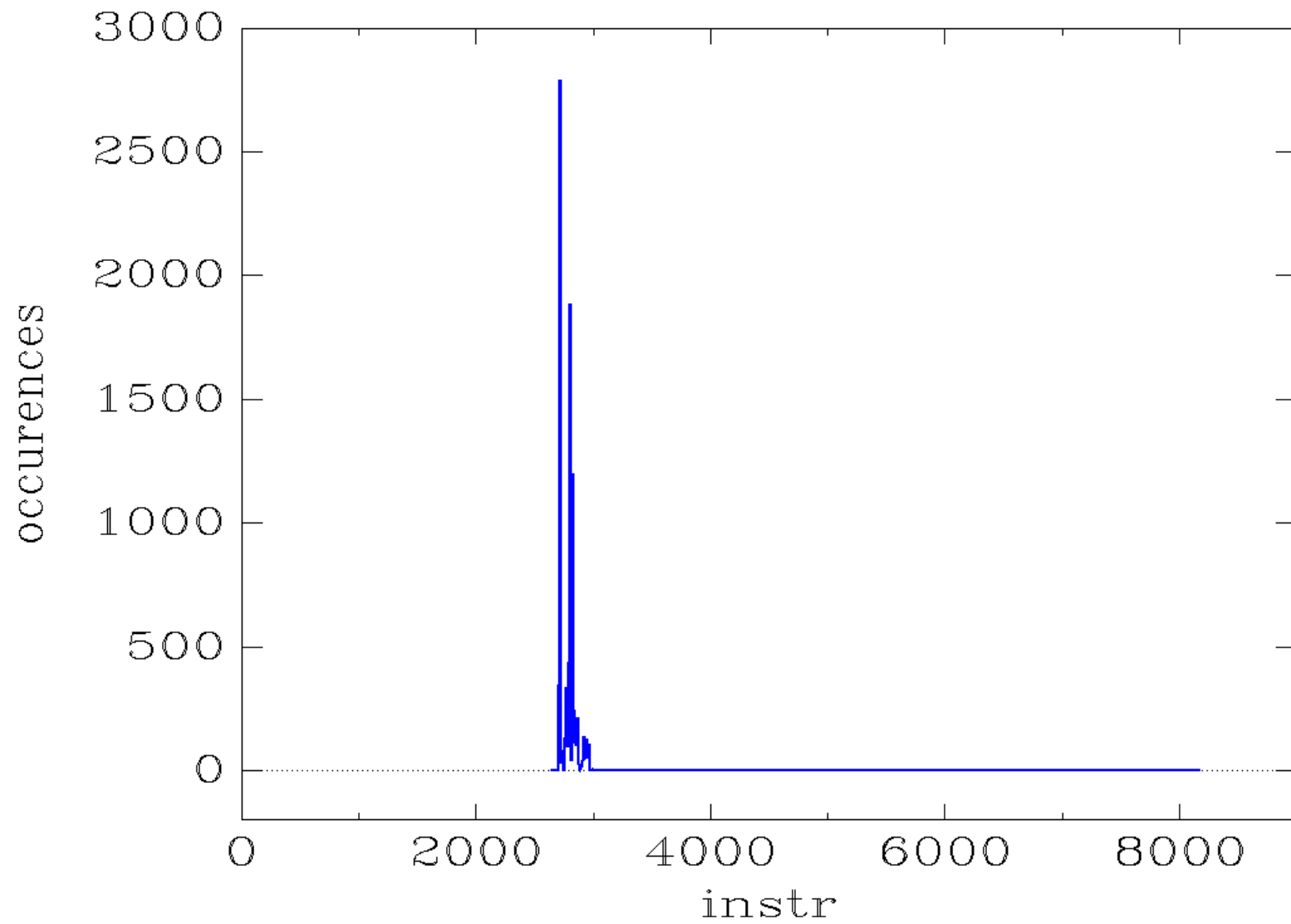
But there is still a tail on the latency graph.

What is causing the tail?

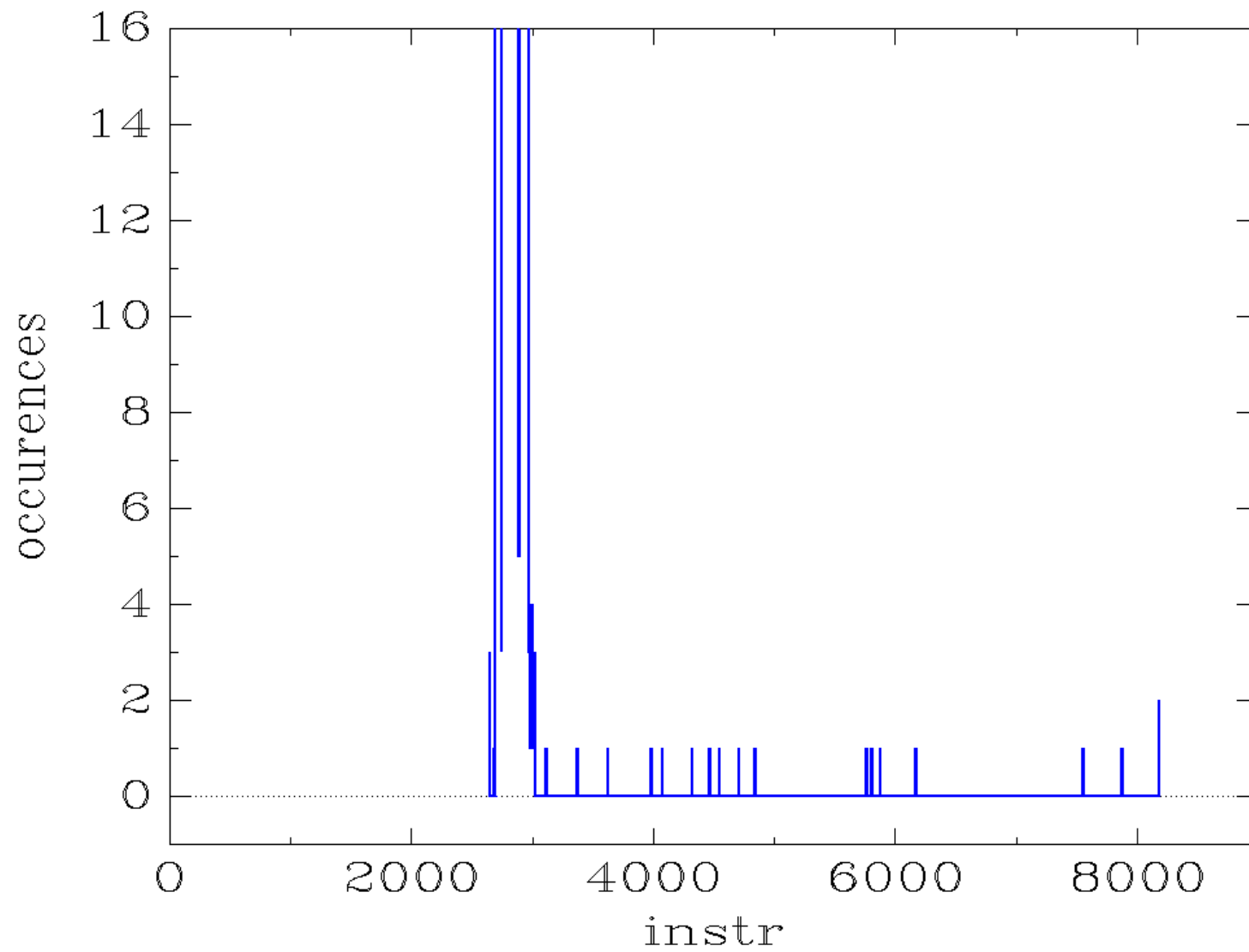
The Well Known Craft may provide some theories, but not a conclusive answer.

Looking at the number of instructions executed tail leads to some new questions.

hist_irqs_off_27_h-4_c-0_1
cpu 0



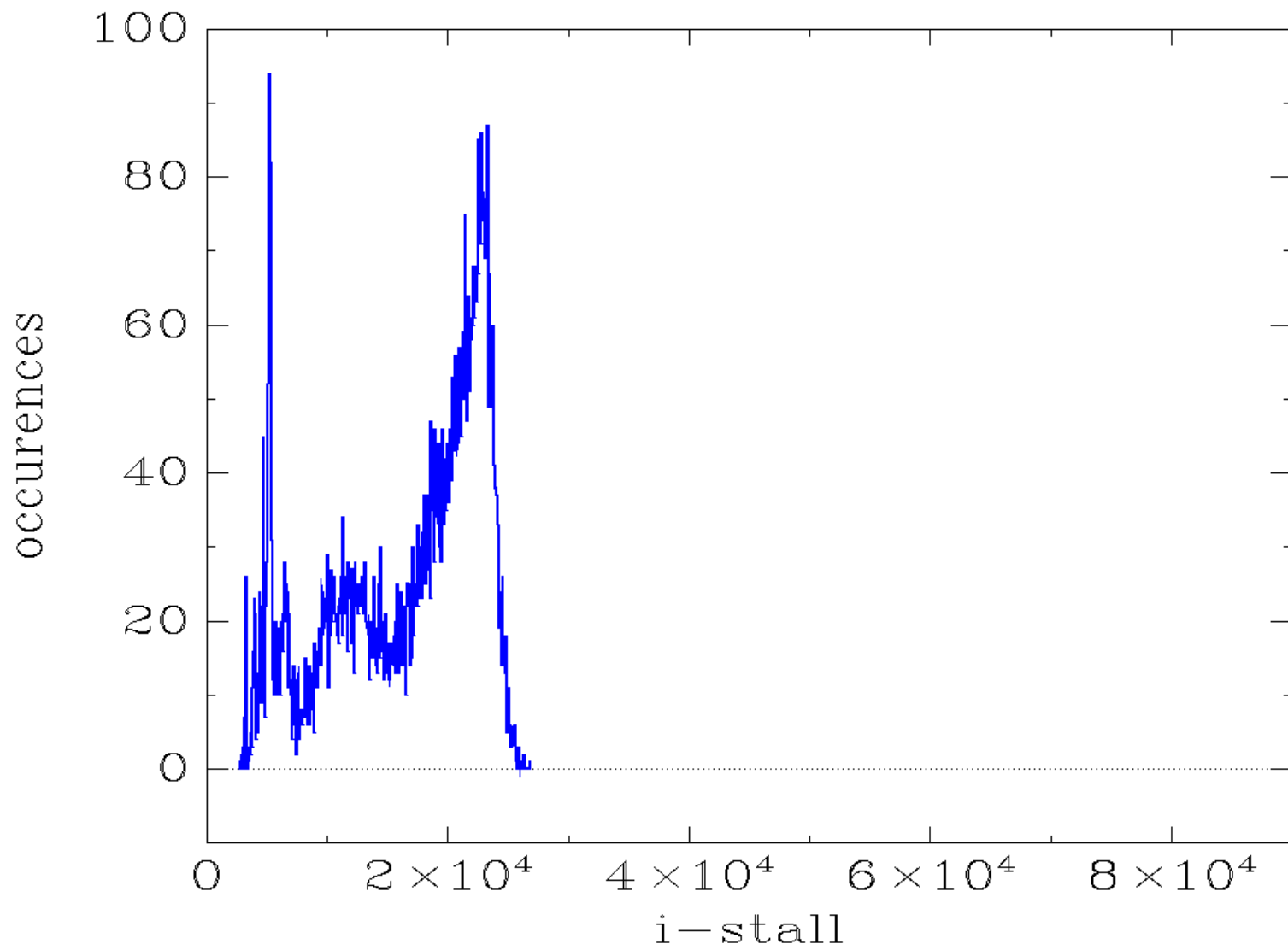
hist_irqs_off_27_h-4_c-0_1
cpu 0



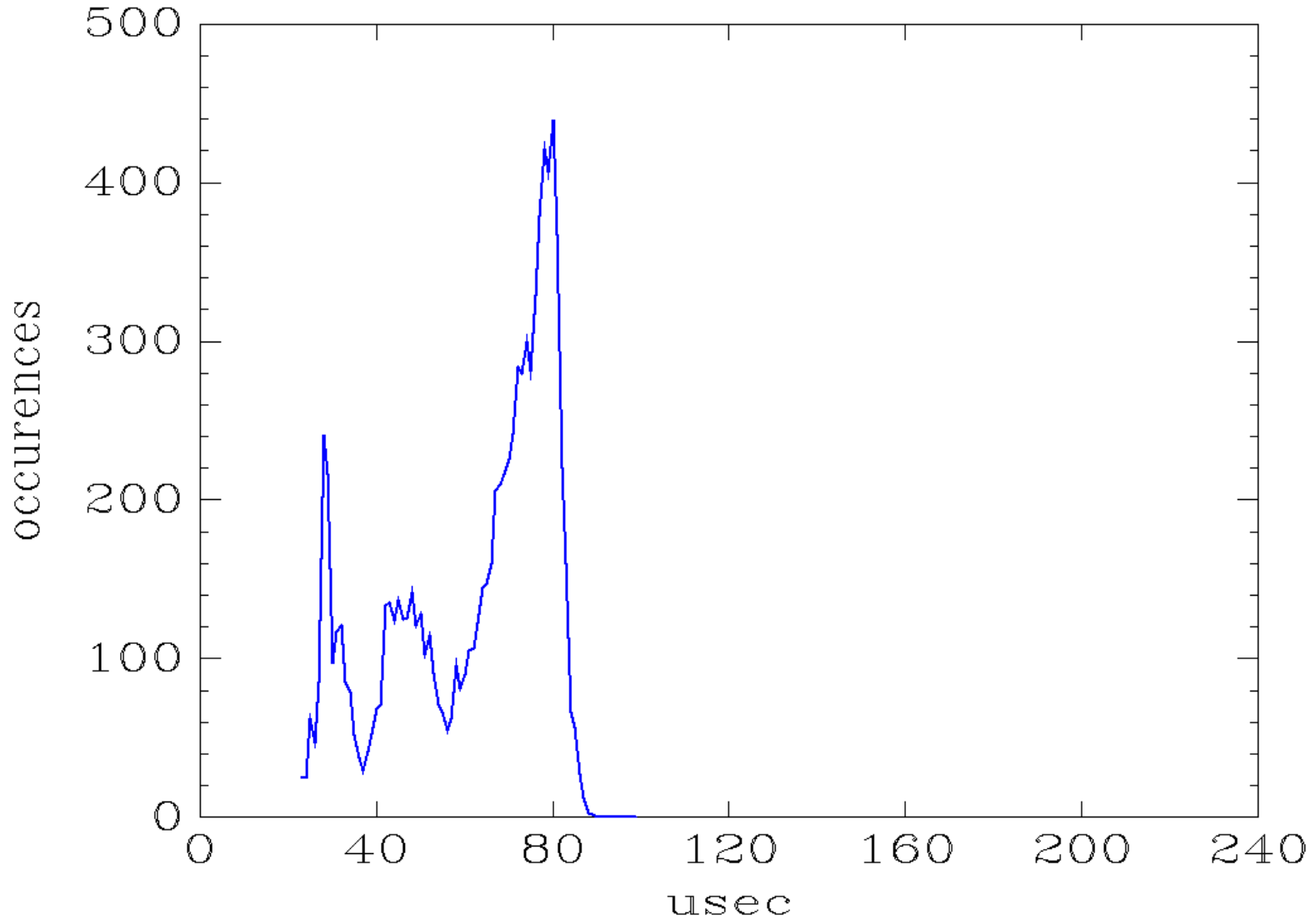
Compare to other tails

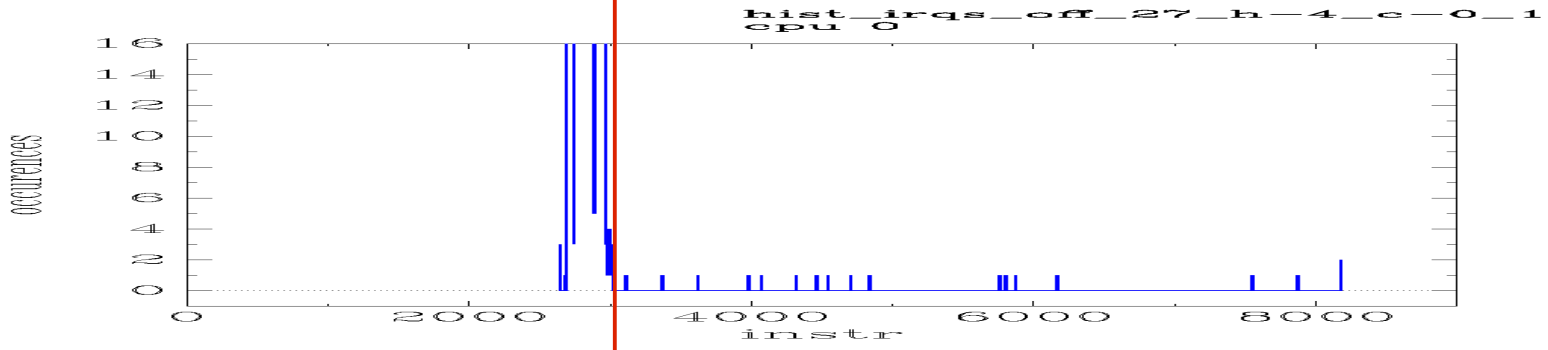
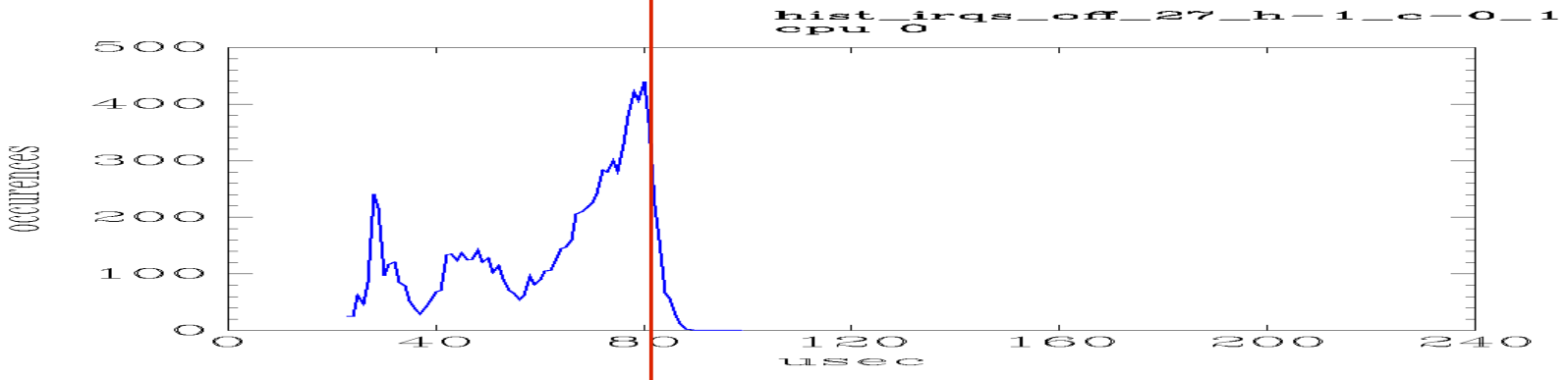
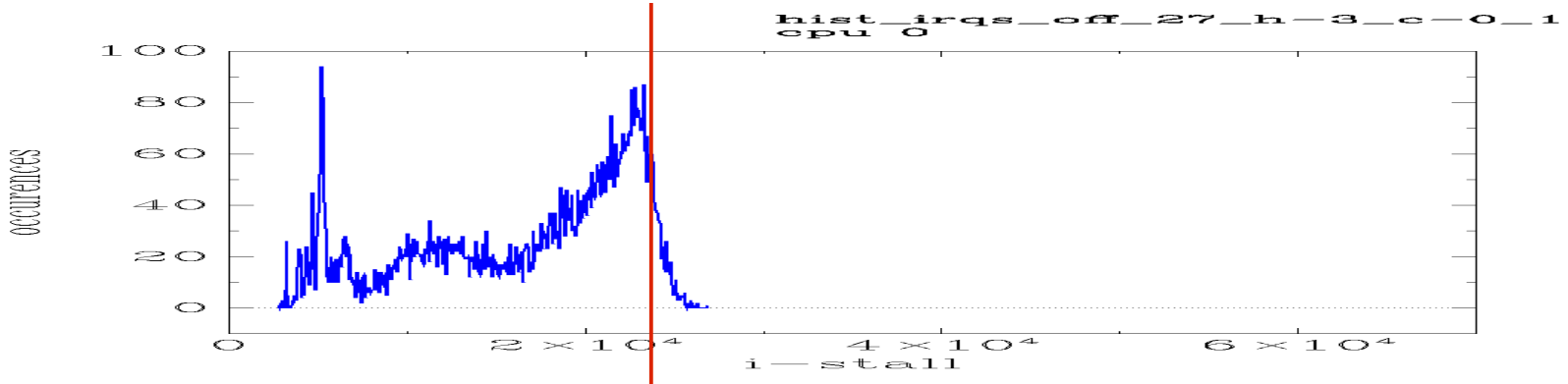
The graphs seen previously, but rescaled to make the peaks coincide with the number of instructions peak.

hist_irqs_off_27_h-3_c-0_1
cpu 0



hist_irqs_off_27_h-1_c-0_1
cpu 0





Number of Instructions Tail

instruction count tail outlier $\approx 2.9 * \text{peak}$

I-cache miss stall tail outlier $\approx 1.2 * \text{peak}$

This suggests that the instruction tail has a low I-cache miss stall cycles.

Number of Instructions Tail

instruction count tail outlier $\approx 2.9 * \text{peak}$

I-cache miss stall tail outlier $\approx 1.2 * \text{peak}$

This suggests that the instruction tail has a low I-cache miss stall cycles.

This is consistent with code spinning in a loop.

Number of Instructions Tail

instruction count tail outlier $\approx 2.9 * \text{peak}$

I-cache miss stall tail outlier $\approx 1.2 * \text{peak}$

This suggests that the instruction tail has a low I-cache miss stall cycles.

This is consistent with code spinning in a loop.

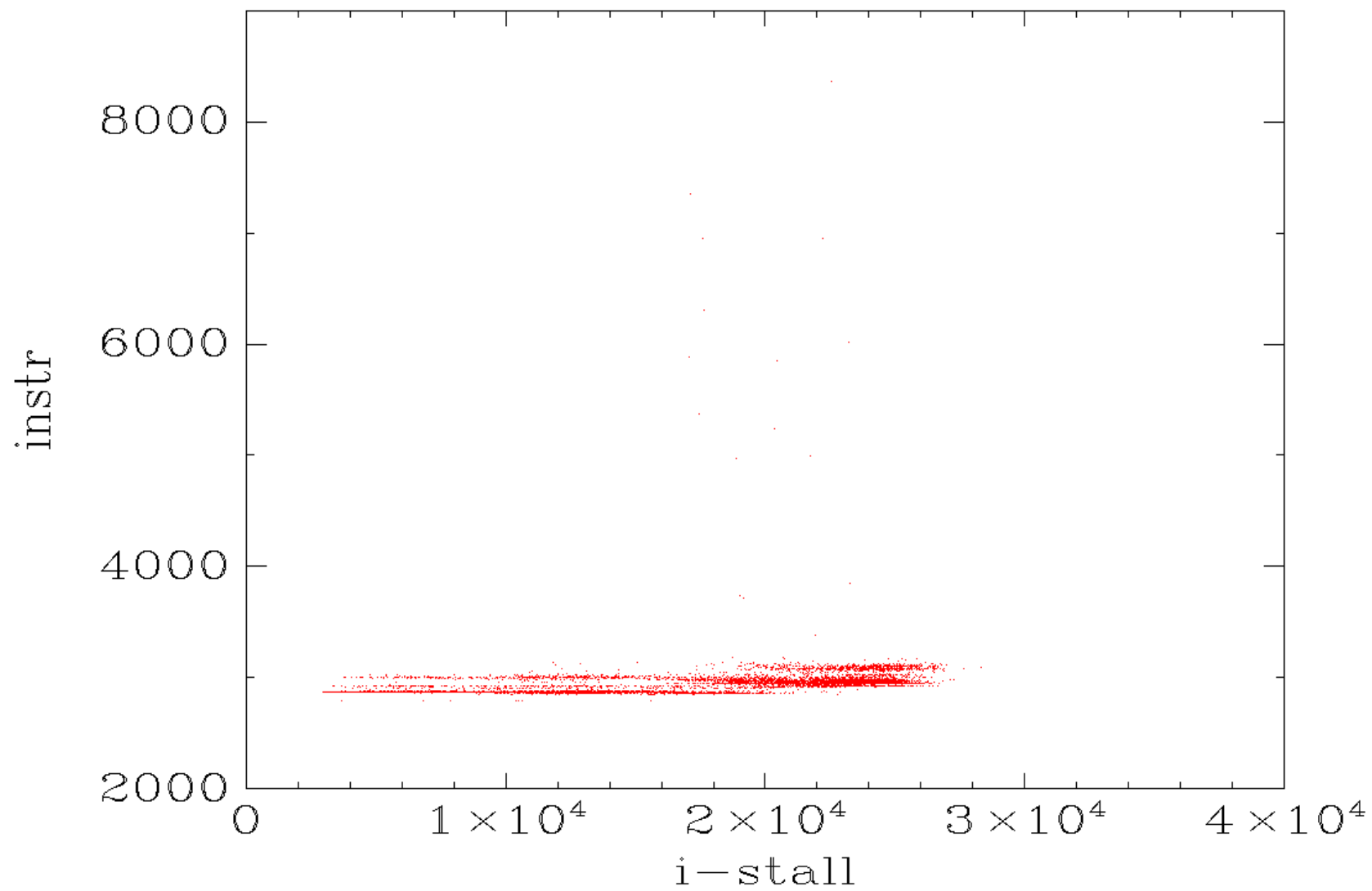
This is consistent with waiting on a spinlock.

Number of Instructions Tail

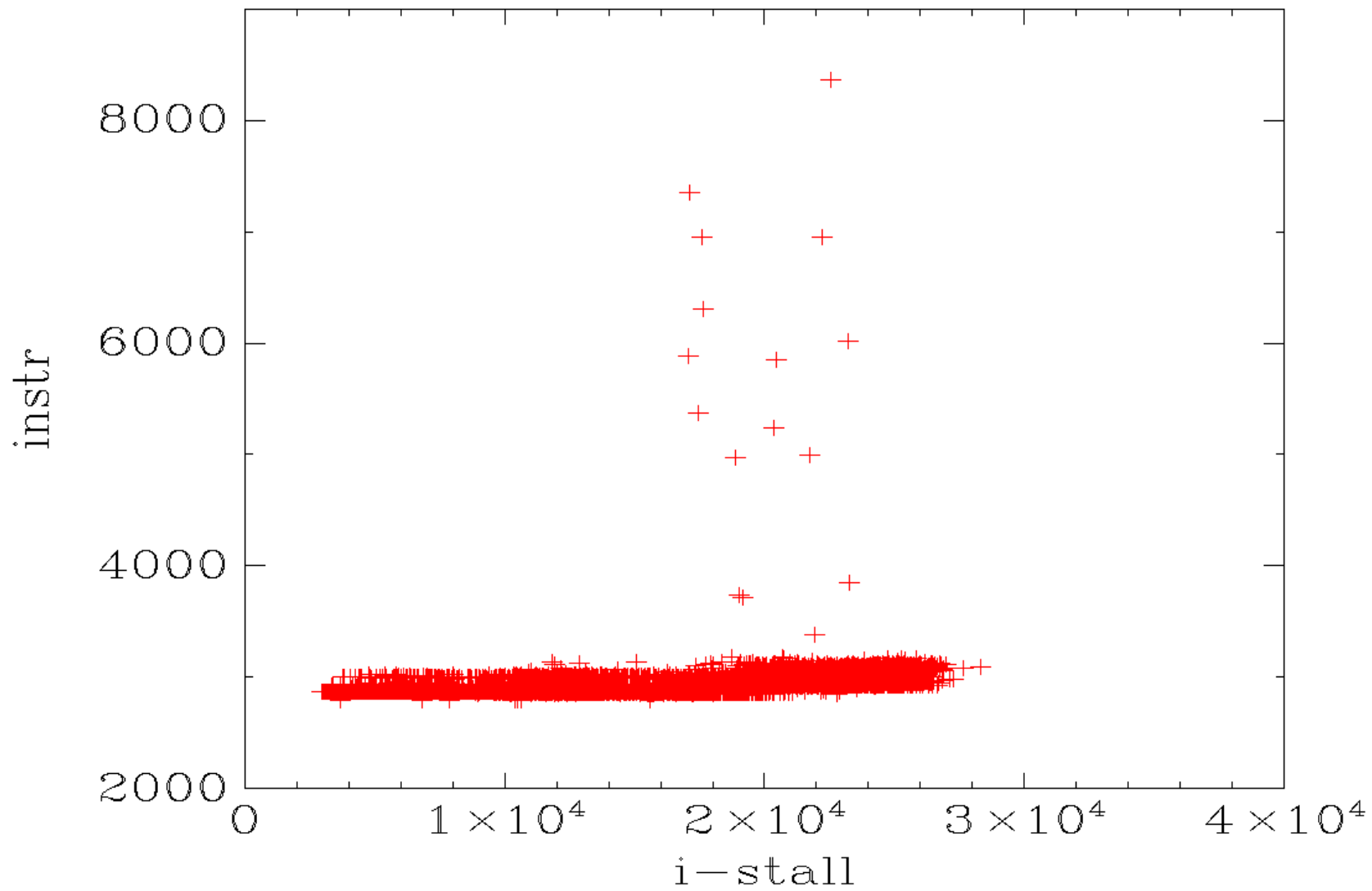
This suggests that the instruction tail has a low I-cache miss stall cycles.

Scattergrams are consistent with this suggestion.

trace_irqs_off_106_f-34_c-0_1_S-0
cpu 0



trace_irqs_off_106_f-34_c-0_1_S-1
cpu 0

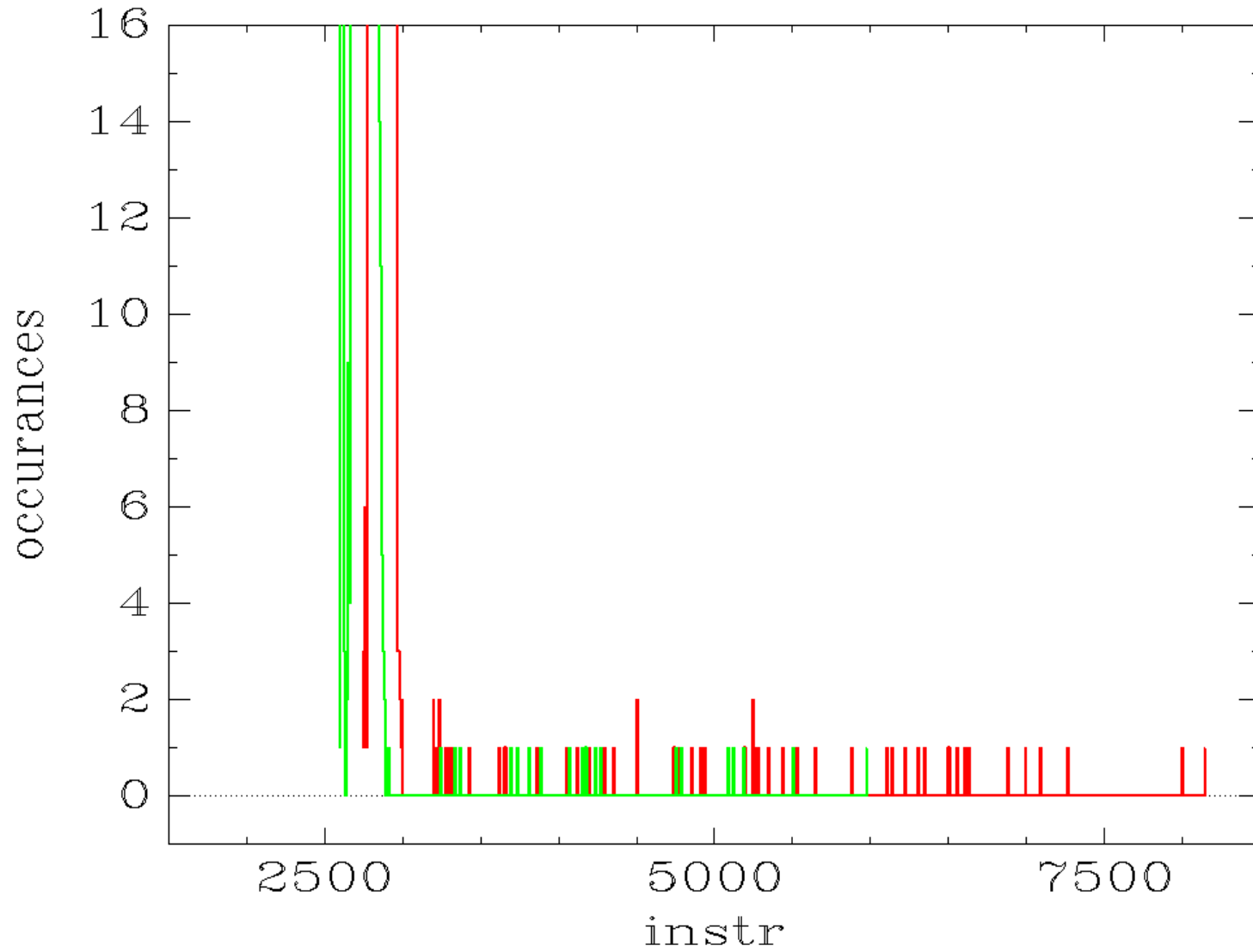


L2 Cache Impact

Adding the L2 cache reduces the magnitude of the number of instructions tail, but does not eliminate it.

hist_irqs_off_{46-45}_h-4_c-0_1

red: Realview, L2 off green: Realview, L2 on



The Well Known Craft (mostly)

/proc/lock_stat

lock_stat version 0.3.2

```
-----  
                -- contention --      ----- wait time -----  
                bounce   count      min     avg     max     total  
-----  
(raw_spinlock_t *)(&rq->lock):      0     94     7.32   14.18   32.22   1333.29  
  |--- cpu 0 :                       0     35     7.32   14.05   32.22   491.73  
  |--- cpu 1 :                       0     59     9.45   14.26   30.09   841.56  
-----  
(raw_spinlock_t *)(&rq->lock)      94     [<c016cbbc>] task_rq_lock+0x38/0x78  
-----  
(raw_spinlock_t *)(&rq->lock)      94     [<c016cbbc>] task_rq_lock+0x38/0x78
```

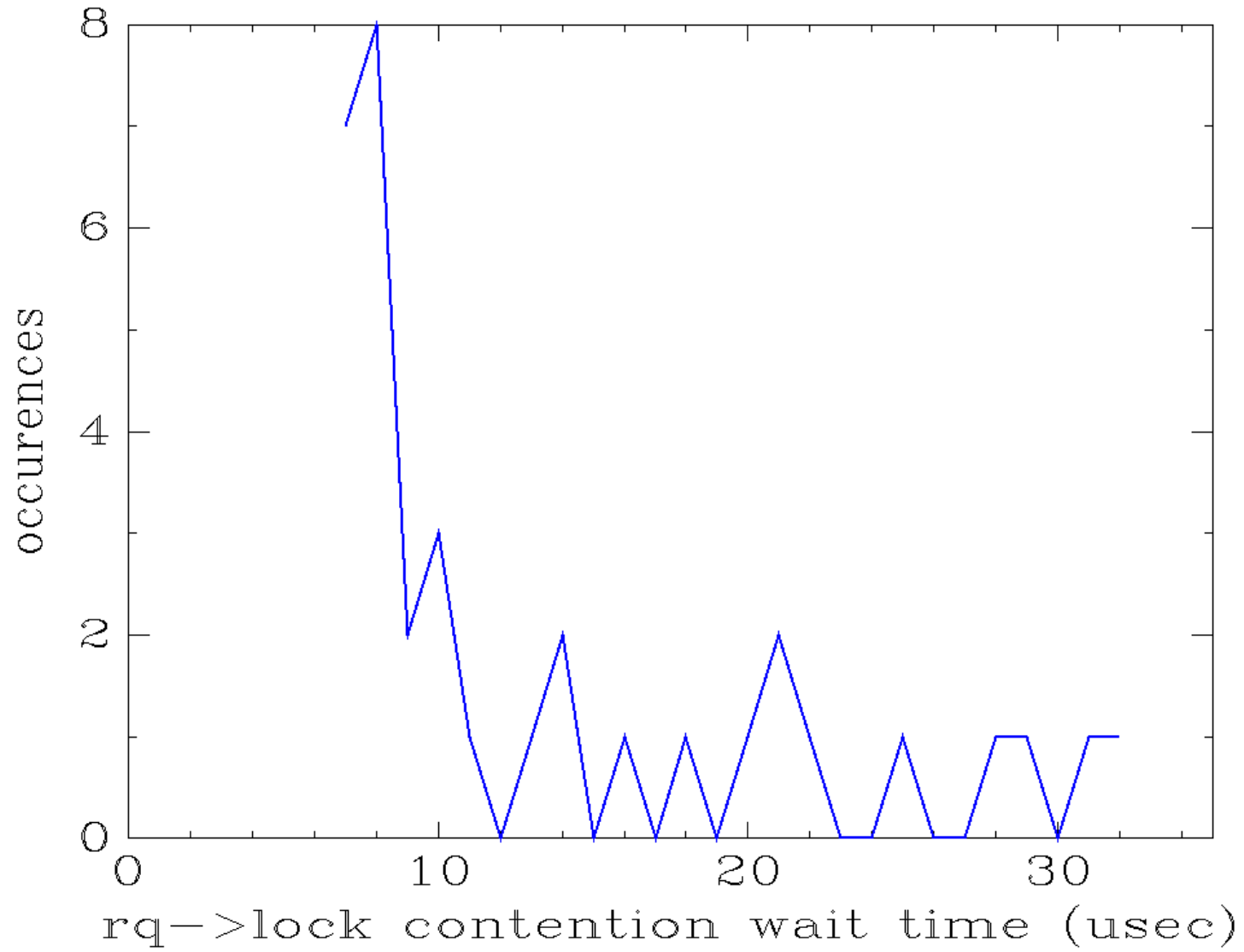
The Well Known Craft (mostly)

`/proc/lock_stat`

But modified to

- only collect data during `do_local_timer()`
- report lock wait and hold time histograms

irqs_off_67 lock_stat_hist_01
cpu 0



Result

The $rq \rightarrow \text{lock}$ wait time tail is consistent with the theory that the number of instructions tail is caused by lock contention. Both tails have a small number of events spread randomly across the tail, with a small number of occurrences of any specific value.

But this is not proof.

Where Is The Lock Attempt?

Determined through existing tools

- /proc/lock_stat
provided the code location of the lock
- ftrace function trace
provided the call graph from do_local_timer()
to the site of the lock

Where Is The Lock Attempt?

Extracted from the ftrace call graph:

```
run_local_timers()
  raise_softirq(TIMER_SOFTIRQ)
    wakeup_softirqd()
      wake_up_process()
        try_to_wake_up()
          task_rq_lock()
            spin_lock_irqsave(&rq->lock, *flags)
```

How To Eliminate Lock Wait?

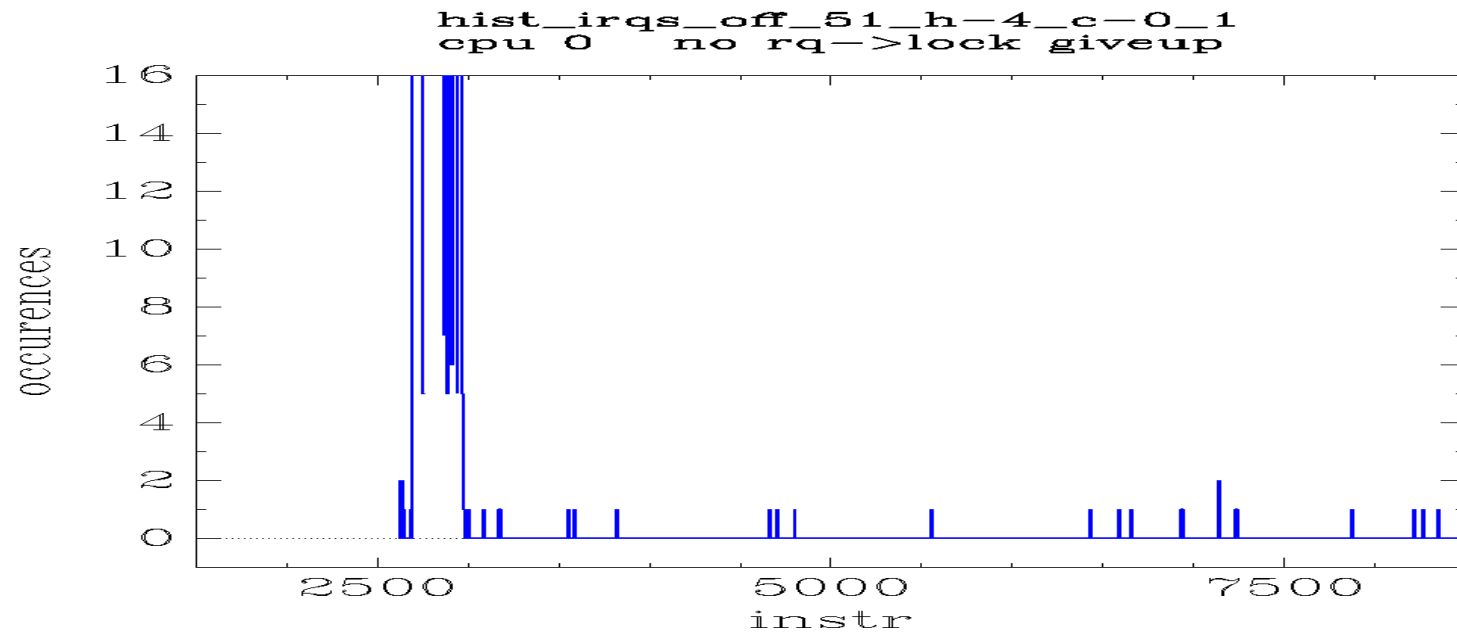
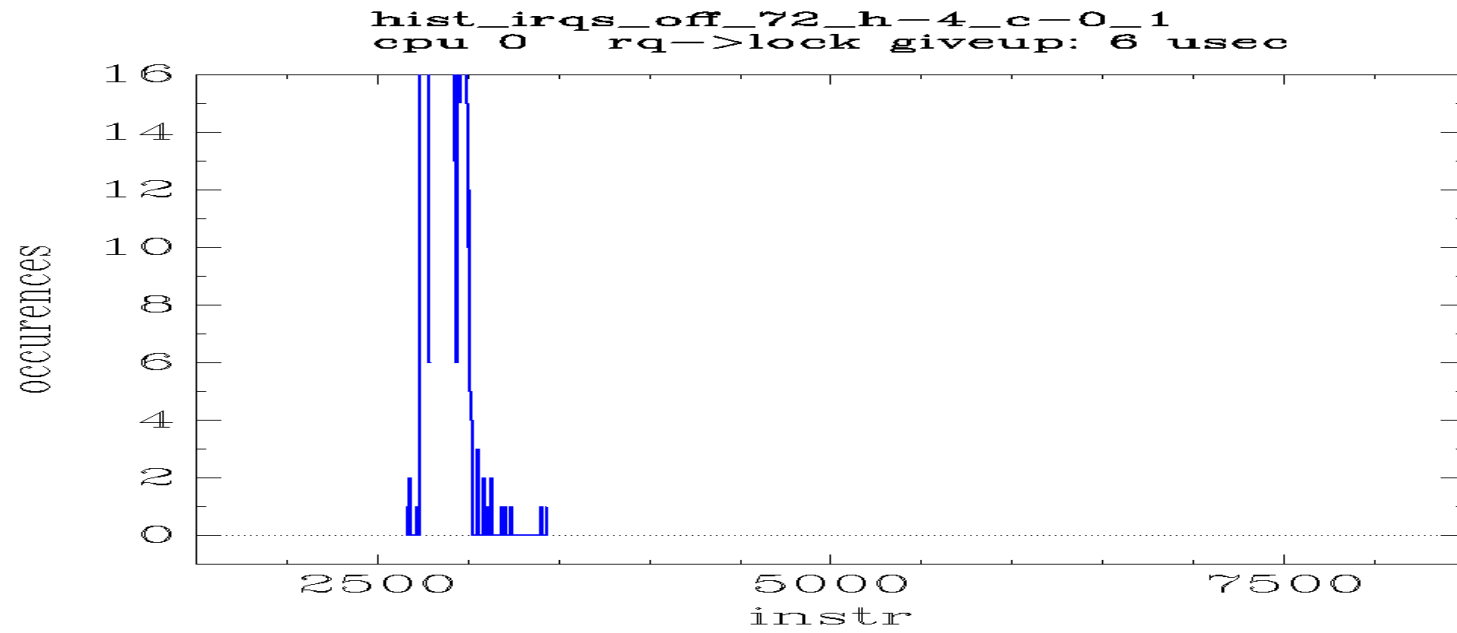
Experiment to determine whether contended `rq->lock` is the cause of the number of instructions tail.

How To Eliminate Lock Wait?

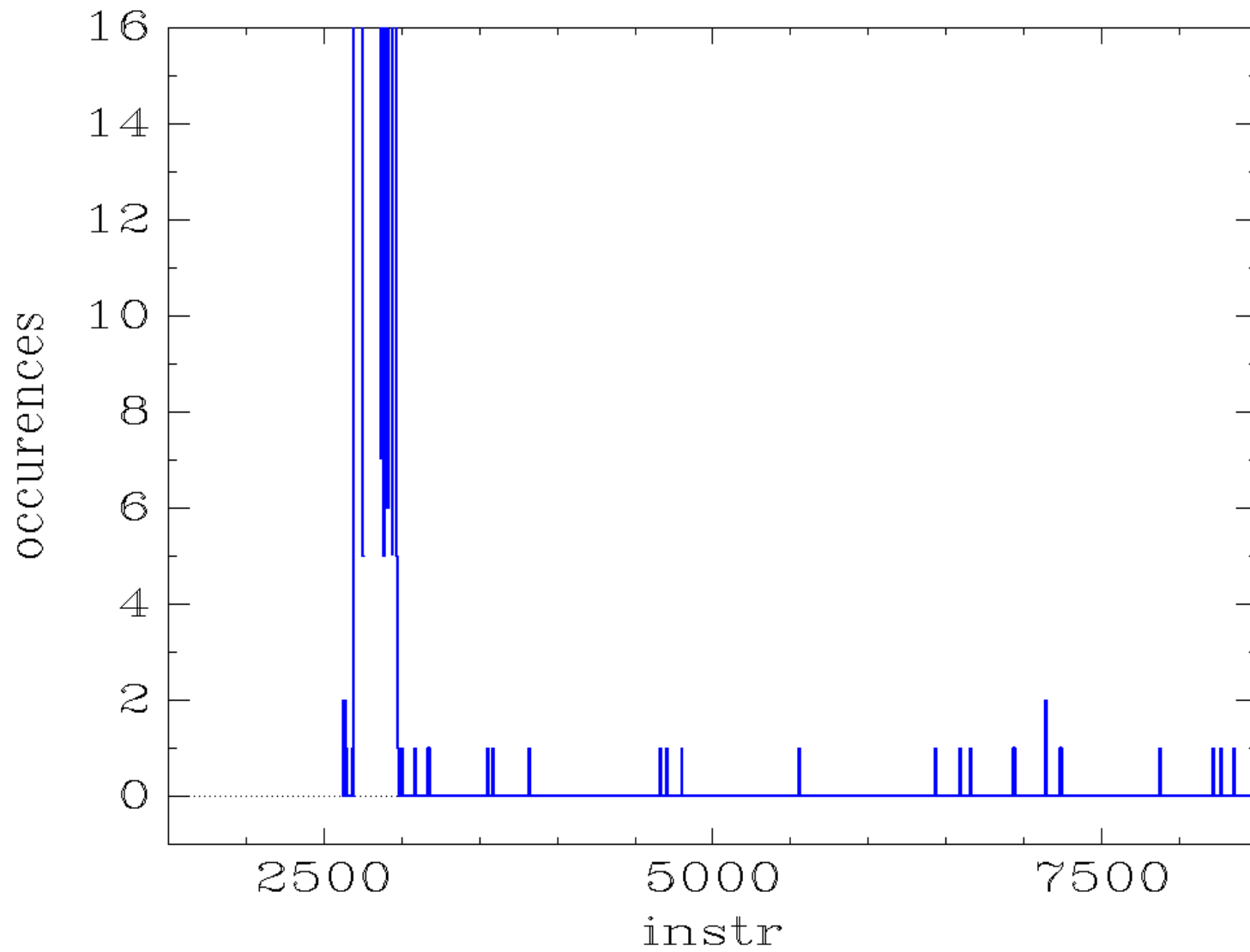
Experiment to determine whether contended `rq->lock` is the cause of the number of instructions tail.

While trying to acquire the `rq->lock`, on this specific code path, give up after a short number of failed attempts. Thus do not wake up the `TIMER_SOFTIRQ` process.

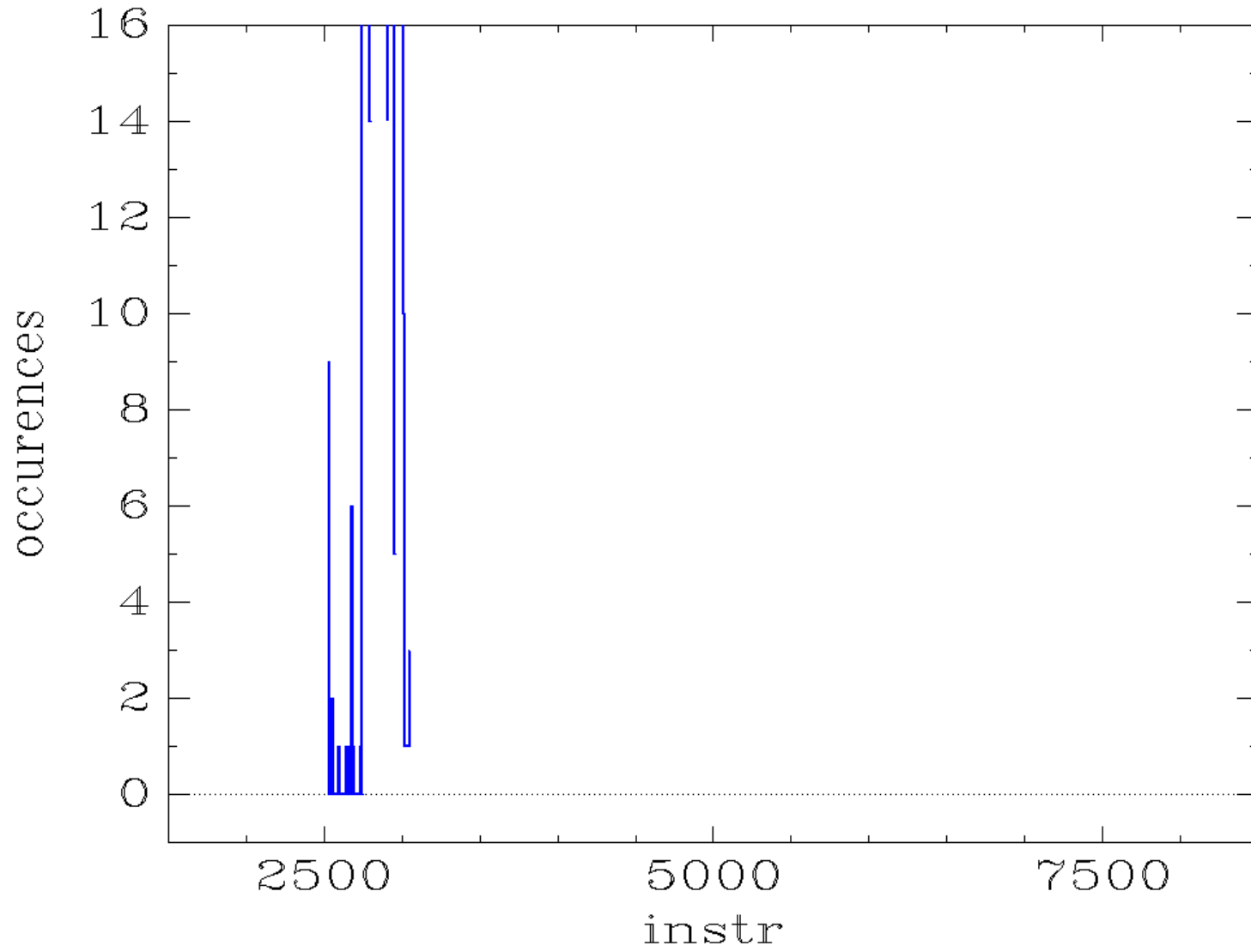
The `TIMER_SOFTIRQ` wakeup will be woken up at the next tick, so some timers will occur late.



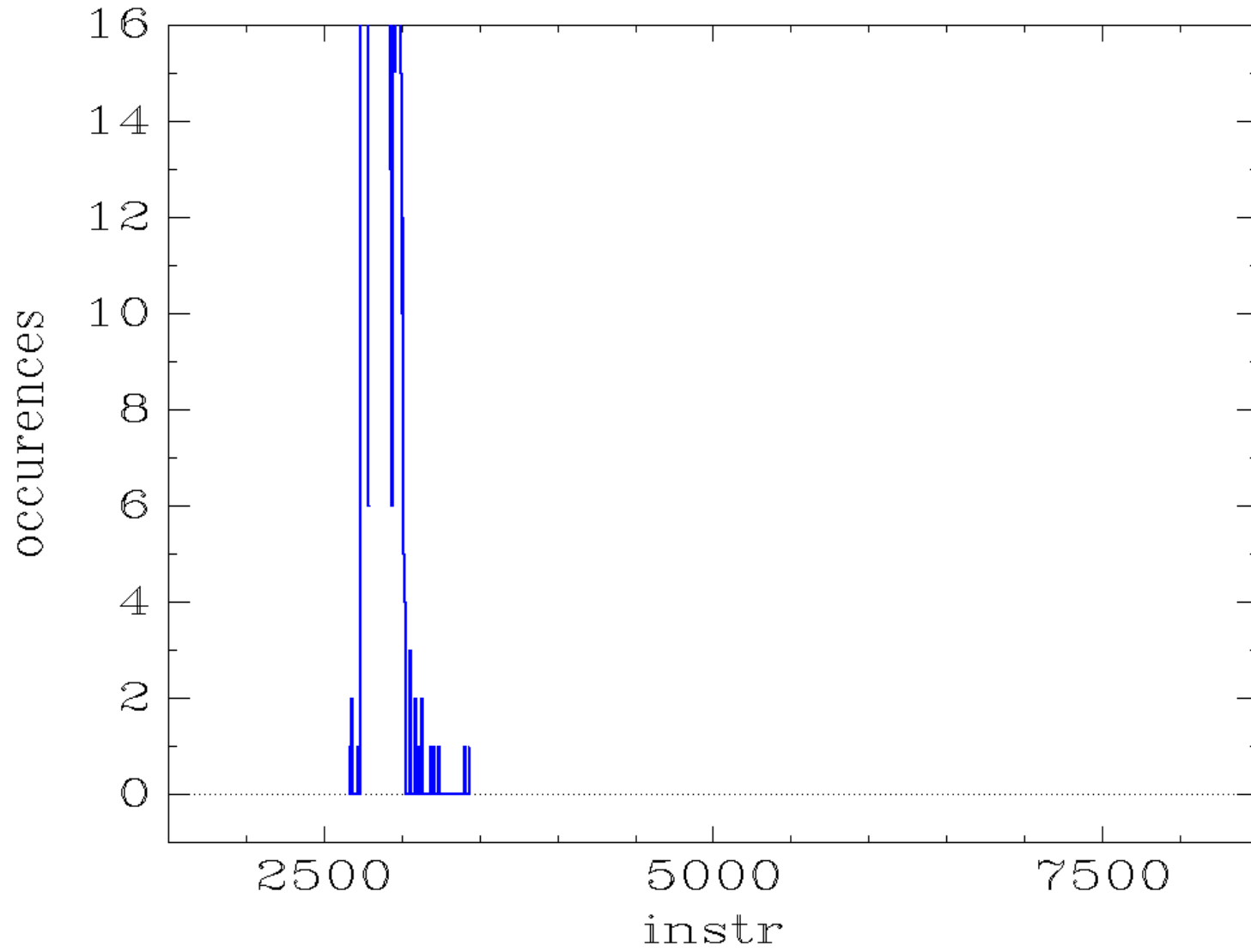
hist_irqs_off_51_h-4_c-0_1
cpu 0 no rq->lock giveup



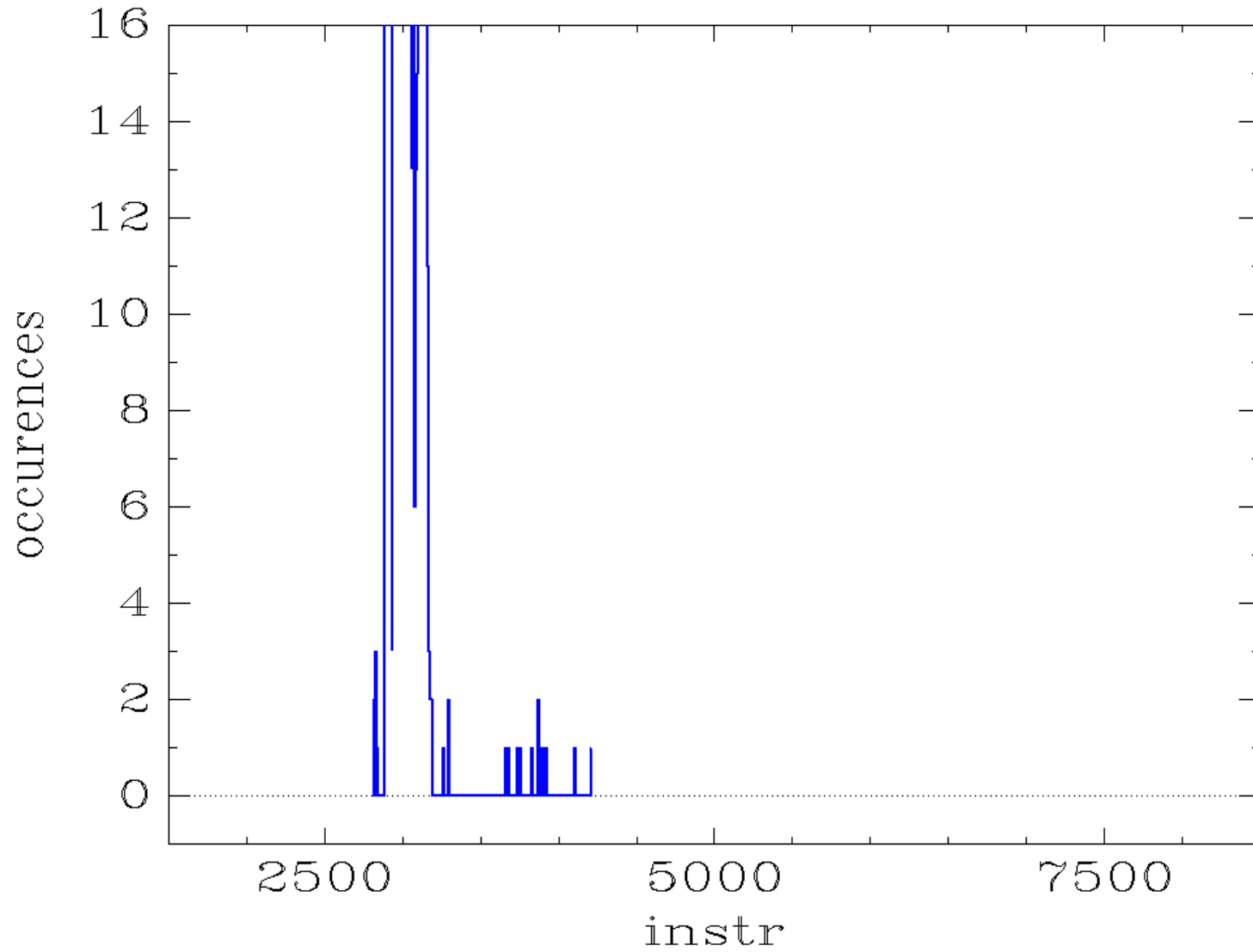
hist_irqs_off_85_h-4_c-0_1
cpu 0 rq->lock giveup: 0 usec



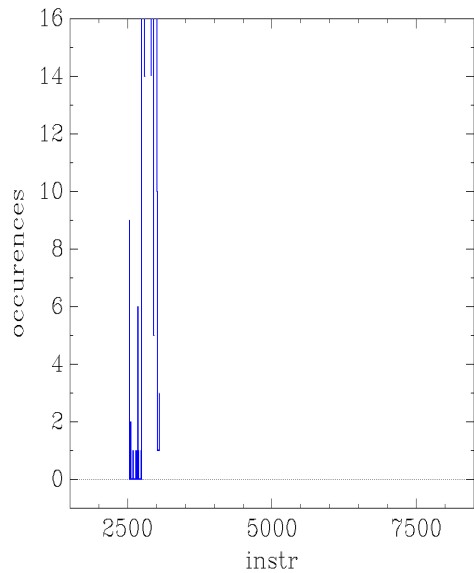
hist_irqs_off_72_h-4_c-0_1
cpu 0 rq->lock giveup: 6 usec



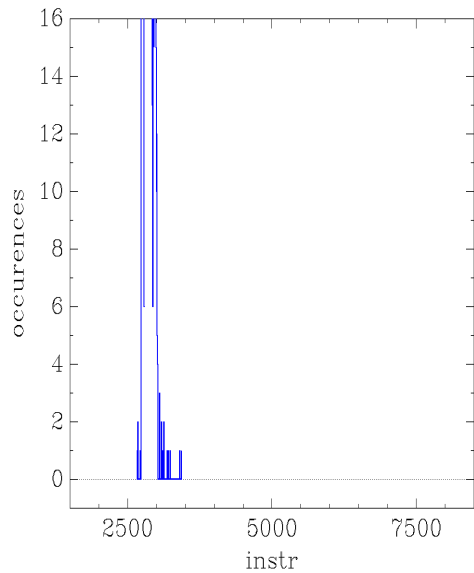
hist_irqs_off_87_h-4_c-0_1
cpu 0 rq->lock giveup: 12 usec



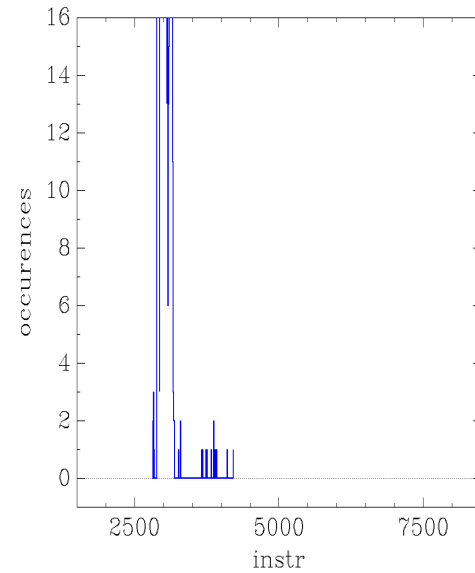
hist_irqs_off_85_h-4_c-0_1
cpu 0 rq->lock giveup: 0 usec



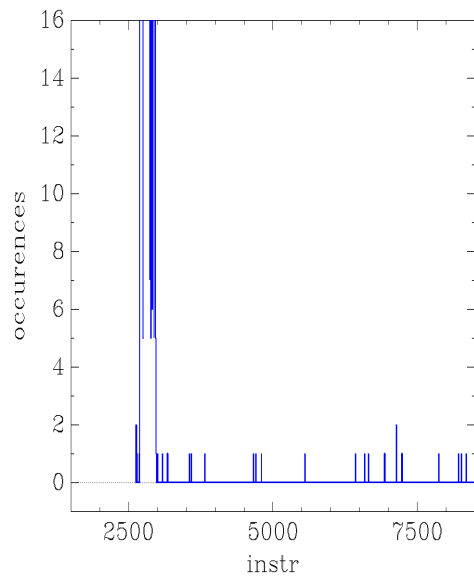
hist_irqs_off_72_h-4_c-0_1
cpu 0 rq->lock giveup: 6 usec



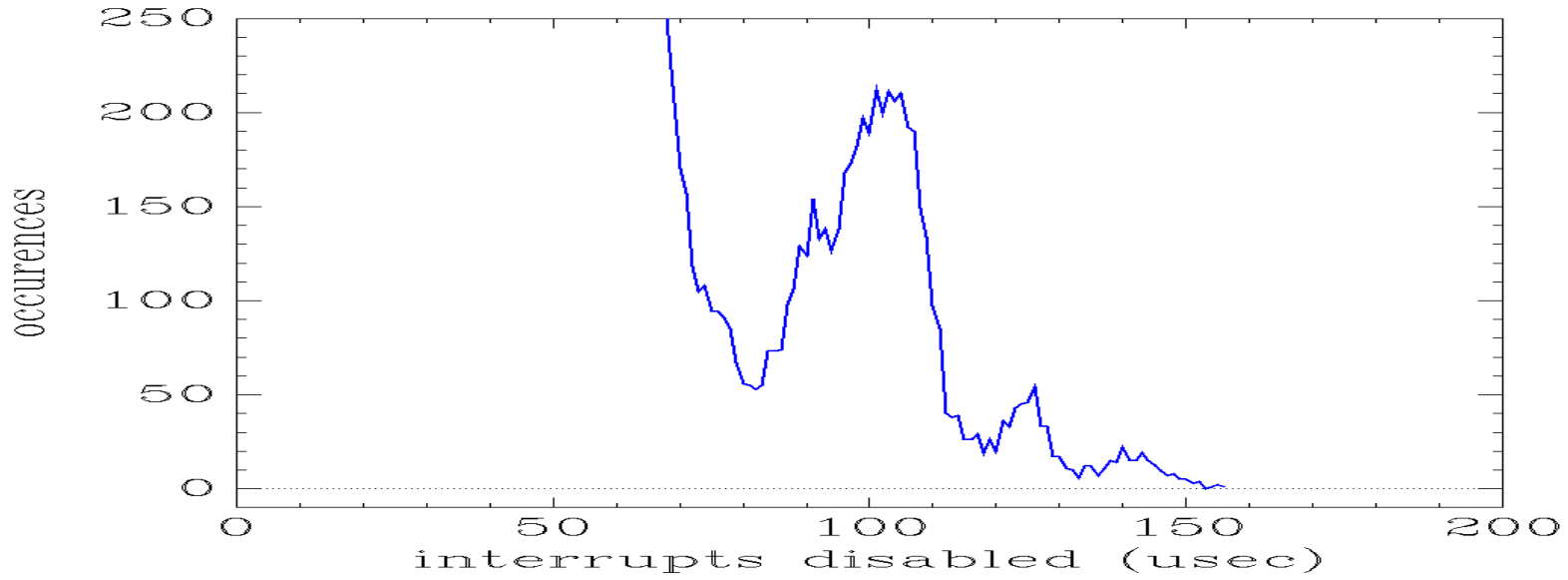
hist_irqs_off_87_h-4_c-0_1
cpu 0 rq->lock giveup: 12 usec



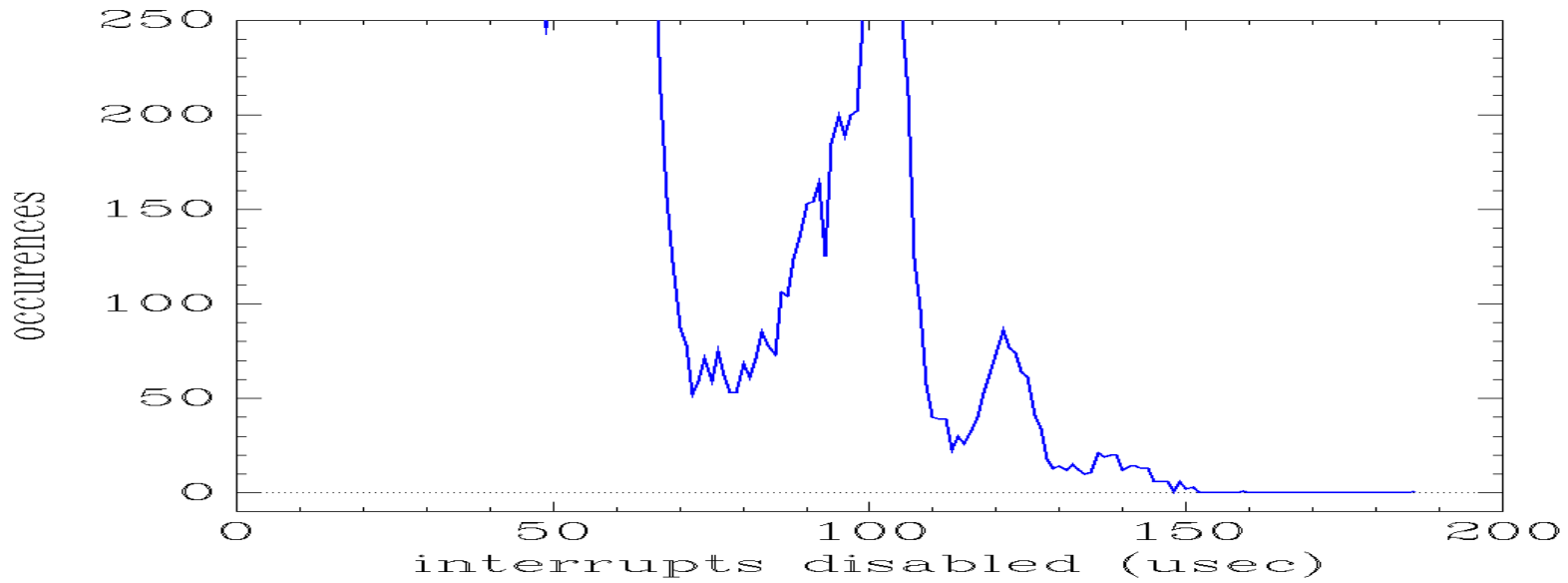
hist_irqs_off_51_h-4_c-0_1
cpu 0 no rq->lock giveup



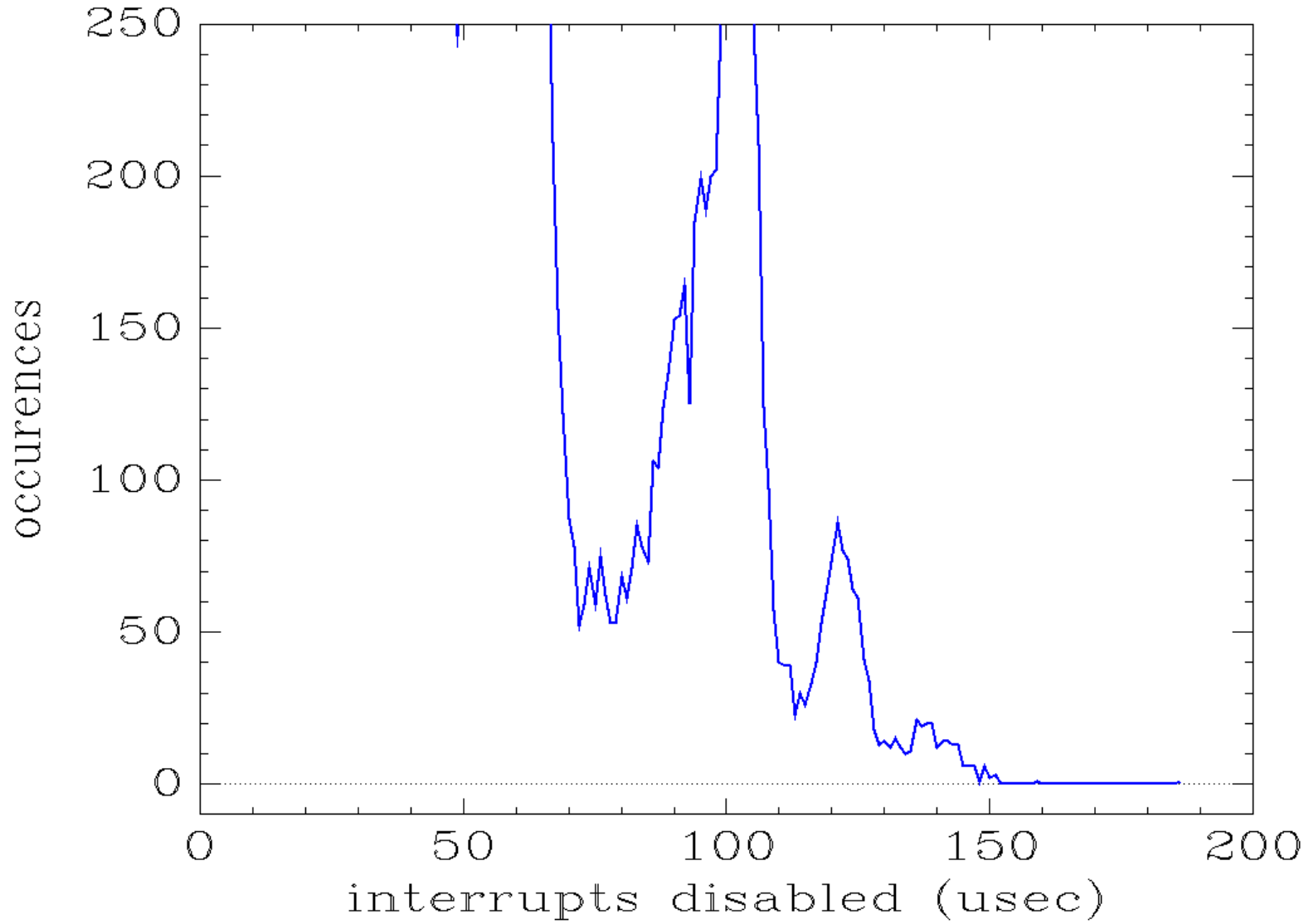
irqs_off_72
cpu 0 rq->lock giveup: 6 usec



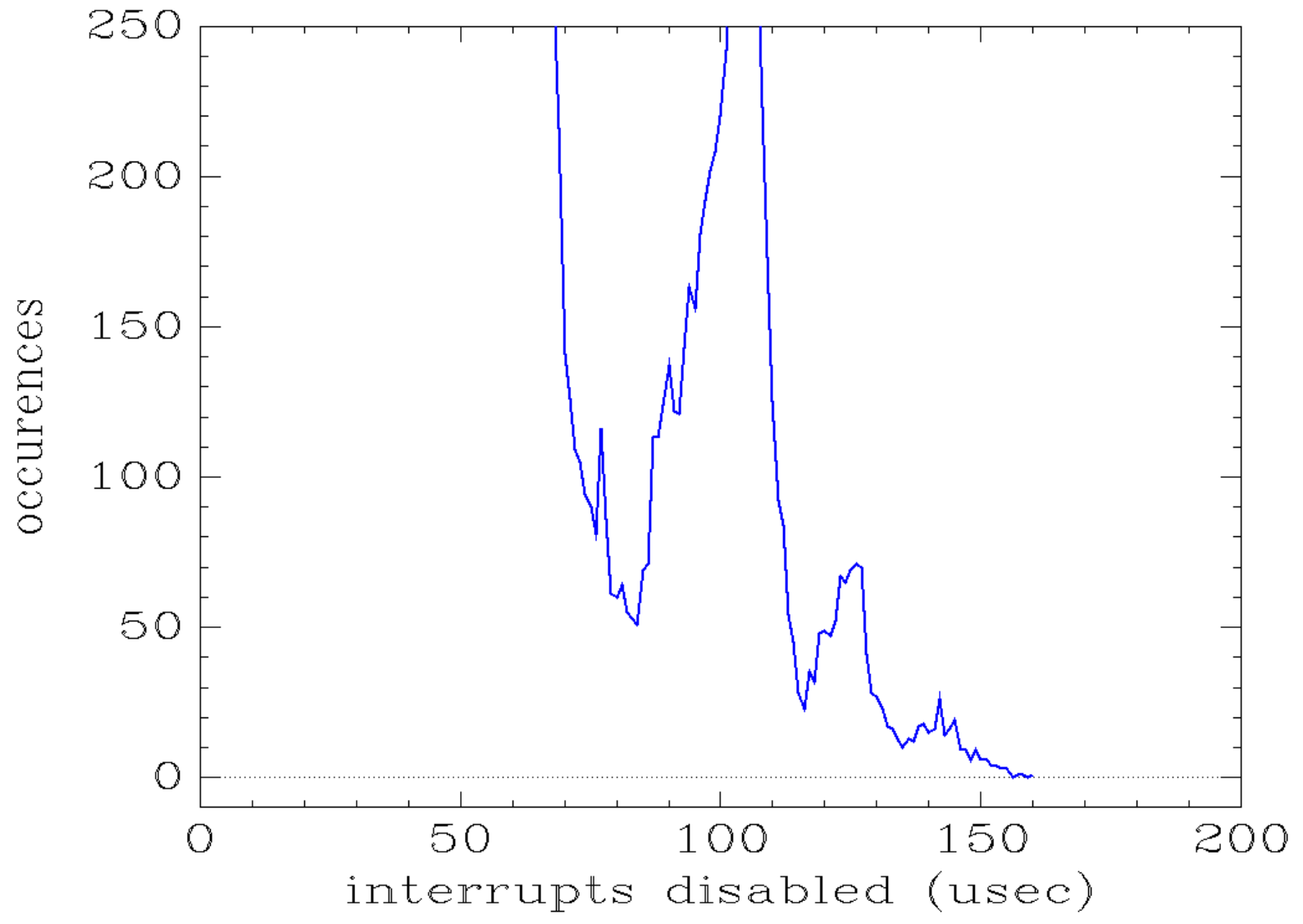
irqs_off_51
cpu 0 no rq->lock giveup



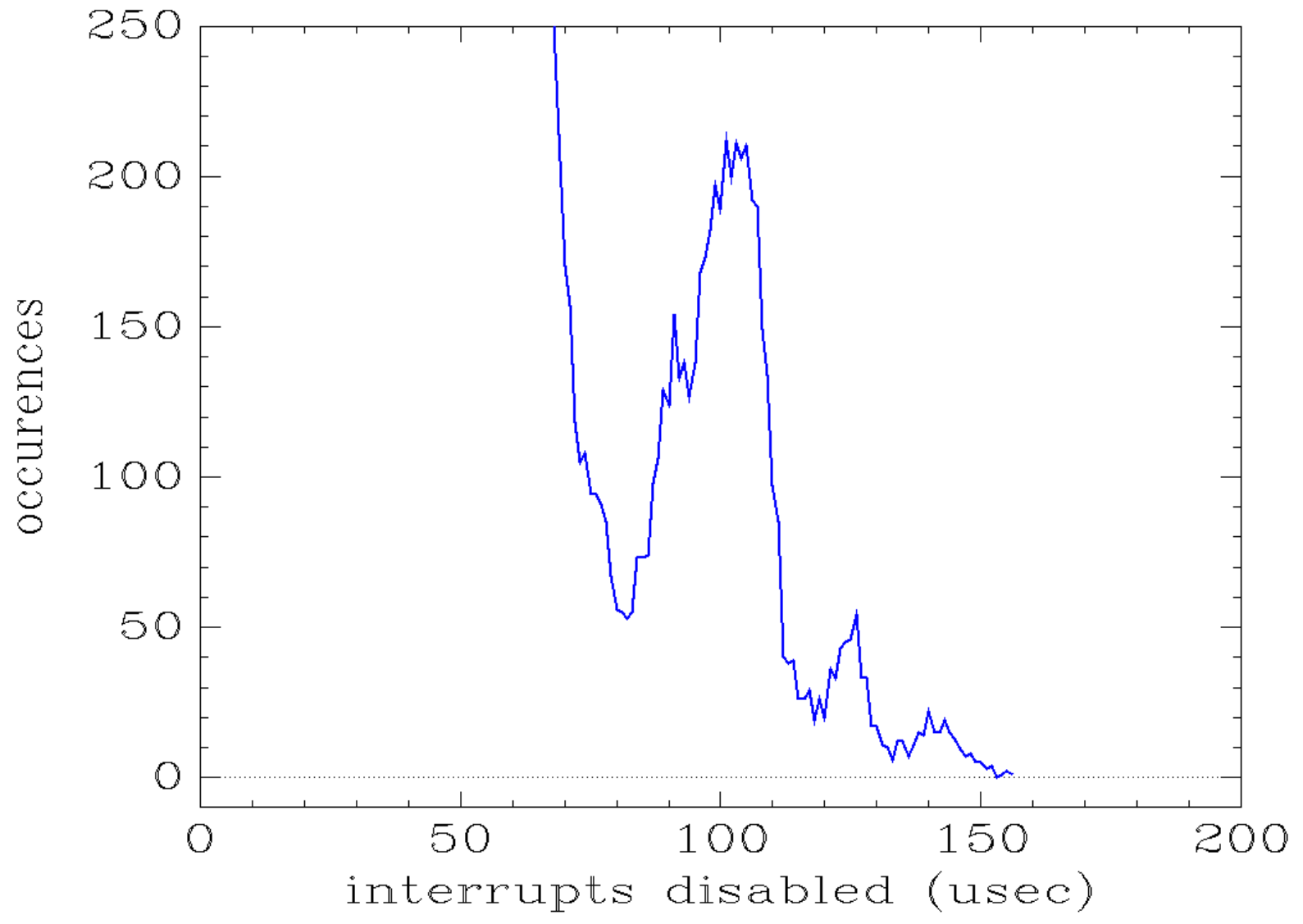
irqs_off_51
cpu 0 no rq->lock giveup



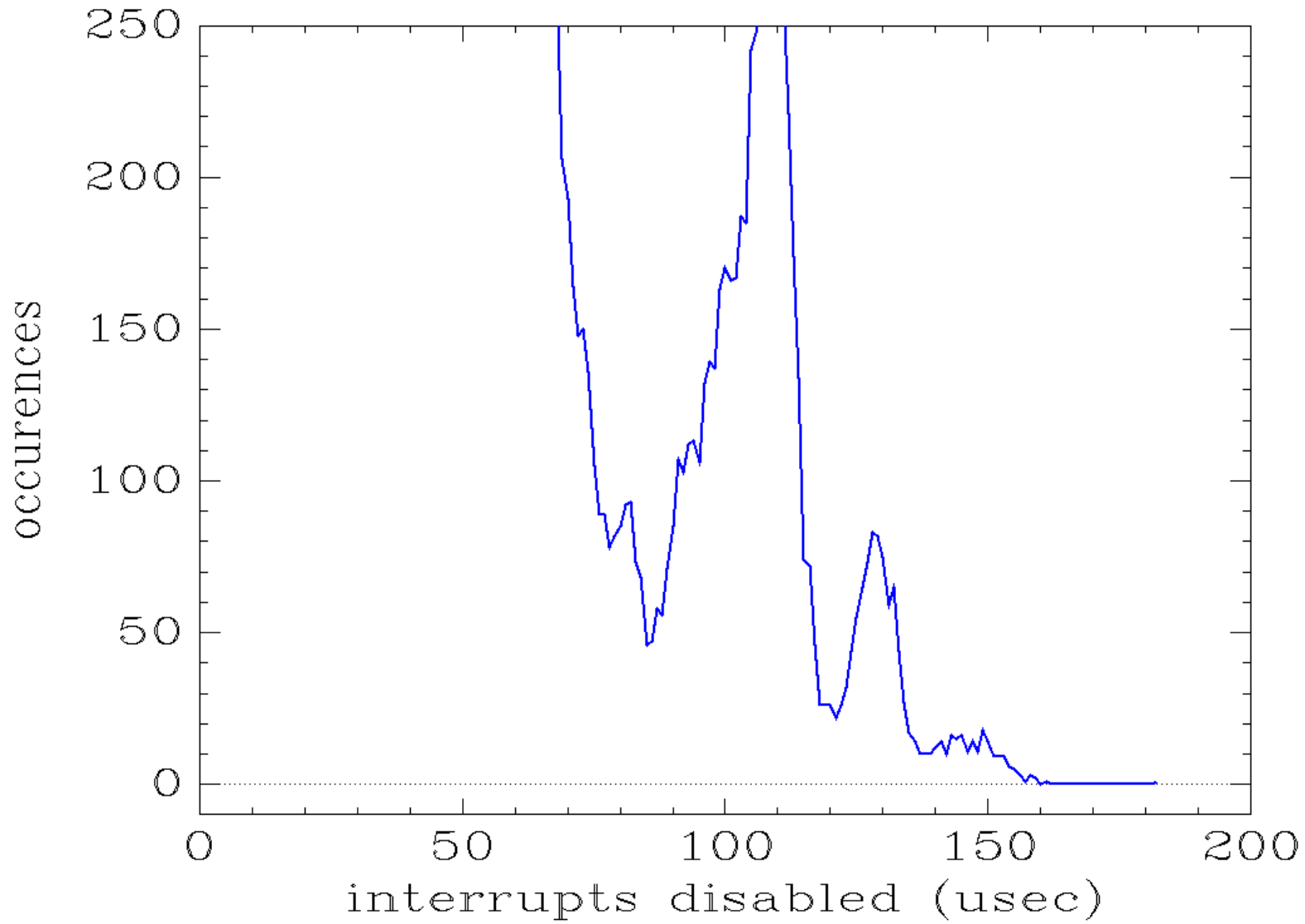
irqs_off_85
cpu 0 rq->lock giveup: 0 usec

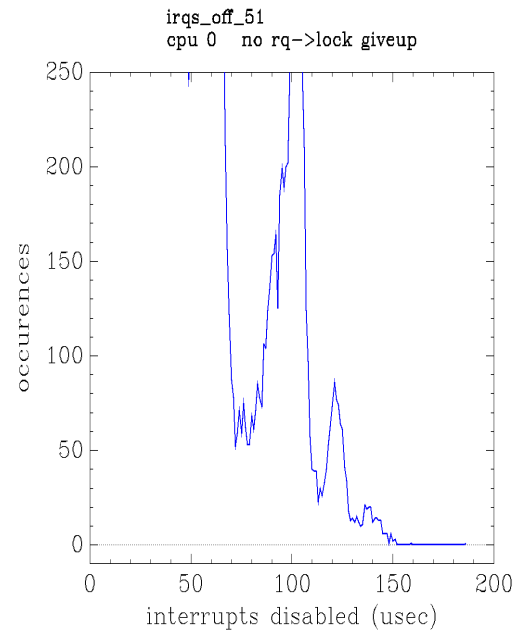
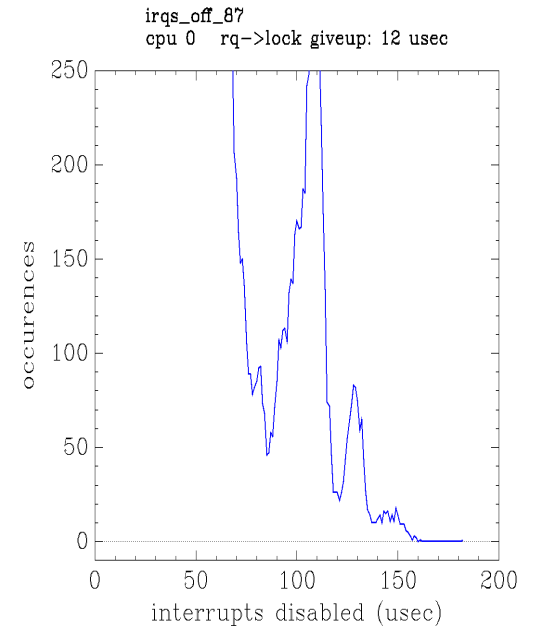
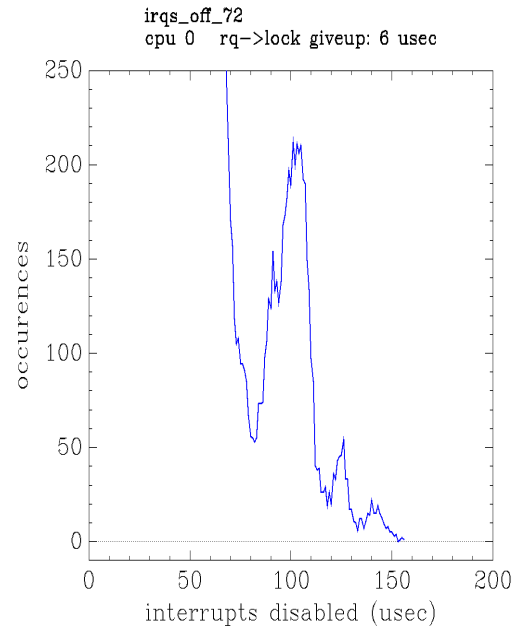
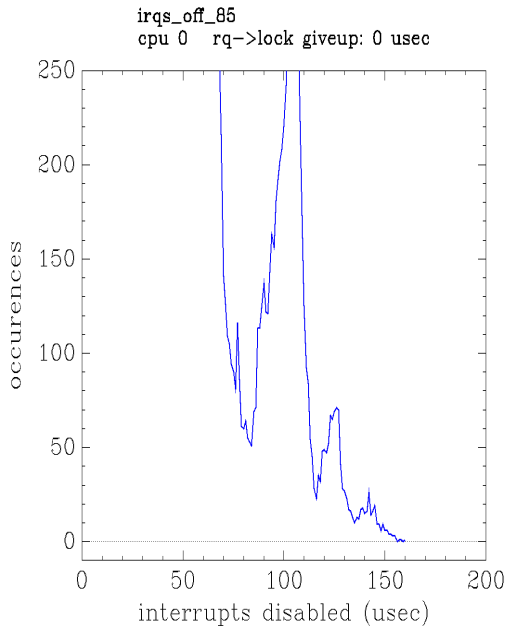


irqs_off_72
cpu 0 rq->lock giveup: 6 usec



irqs_off_87
cpu 0 rq->lock giveup: 12 usec





Result

Reducing lock contention greatly shrank the number of instructions executed tail.

The experimental implementation is not likely to be accepted in mainline, but shows that reducing rq->lock contention can reduce maximum **interrupts disabled duration** in some scenarios.

The correct solution is likely an enhancement of the scheduler locking algorithms.

Reduce runqueue lock contention

LKML

[PATCH RFC] reduce runqueue lock contention

Date: Thu, 20 May 2010 13:48:10 -0700

From: Chris Mason

This is more of a starting point than a patch, but it is something I've been meaning to look at for a long time. Many different workloads end up hammering very hard on `try_to_wake_up`, to the point where the runqueue locks dominate CPU profiles.

Reduce runqueue lock contention

LKML

Re: [PATCH RFC] reduce runqueue lock contention

Date: Mon, 21 Jun 2010 03:02:02 -0700

Peter Zijlstra responded with a competing patch.

Ingo Molnar replied:

“... which future kernel do you aim it for? I'd prefer v2.6.50 or later ;-)

This is a truly scary patch.”

Reduce runqueue lock contention

Peter replied:

“I'd say at the soonest somewhere near
.38/.39 or so.

Its definitely not something we should rush in.”

Reduce runqueue lock contention

LKML

sched: Reduce runqueue lock contention -v6

Date: Tue, 5 Apr 2011 08:23:38 -0700

From: Peter Zijlstra

unpatched: run time 30 seconds 647278 worker burns per second

patched: run time 30 seconds 816715 worker burns per second

I've queued this series for .40.

Obviously bigger is better :-), the above means 26% more wakeups processed in the 30 seconds.

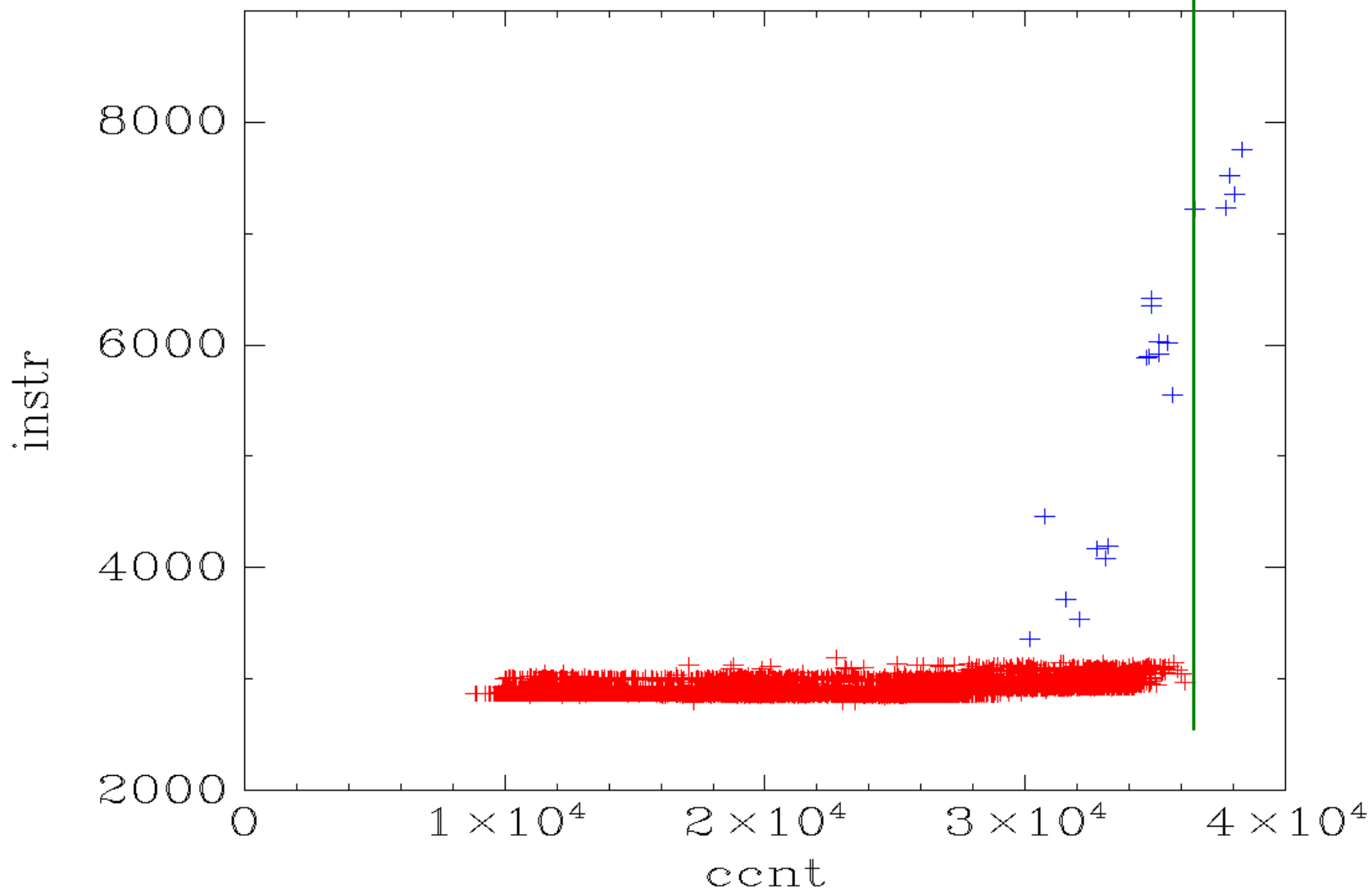
Implications

Does eliminating or reducing the magnitude of the number of instruction tail reduce the maximum **interrupts disabled duration**?

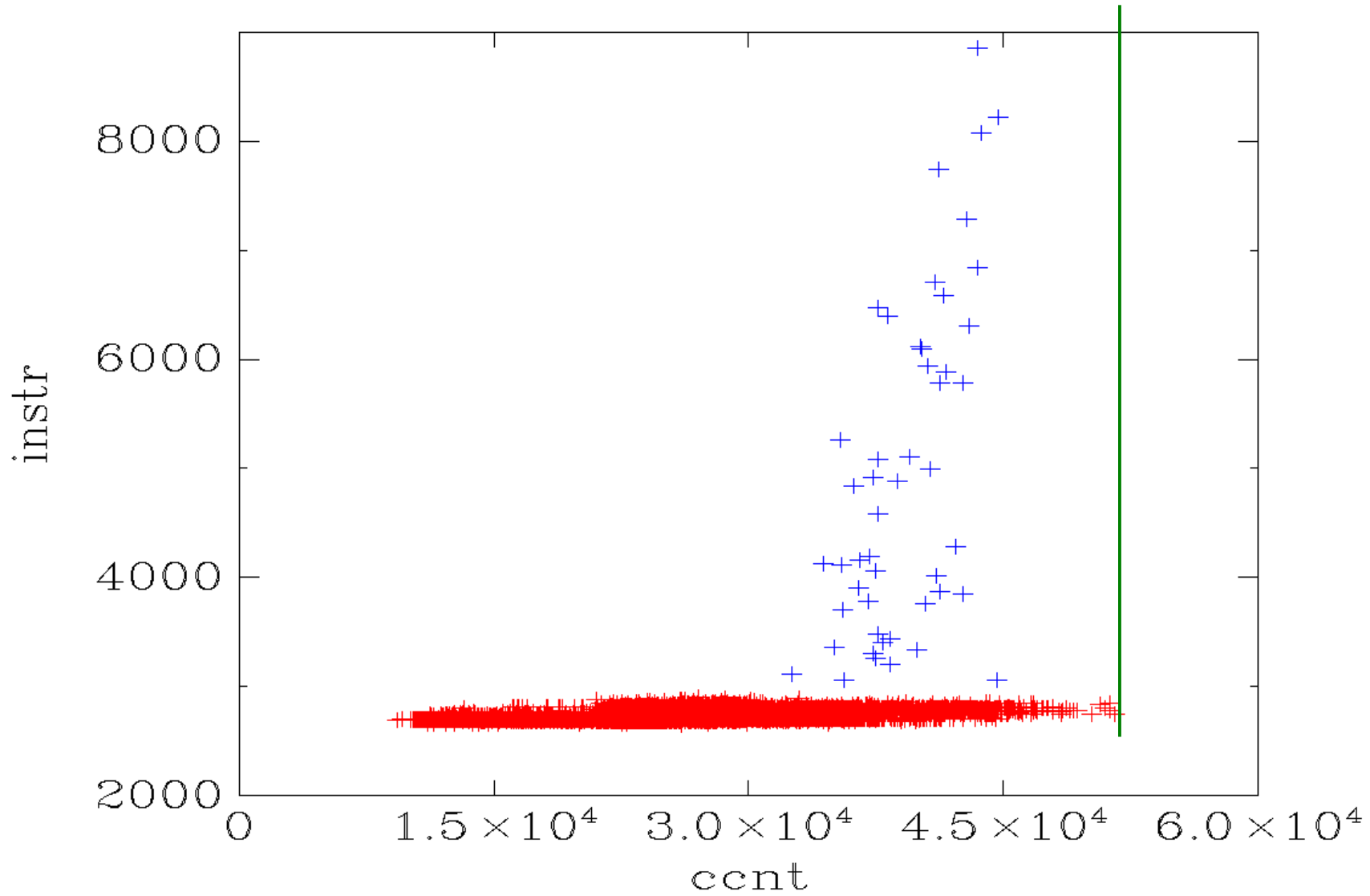
On some test boards: YES

On other test boards: NO

irqs_off_106 test_run 3
cpu 0 Navi



irqs_off_118 test_run 1
cpu 0 Realview L2 disabled



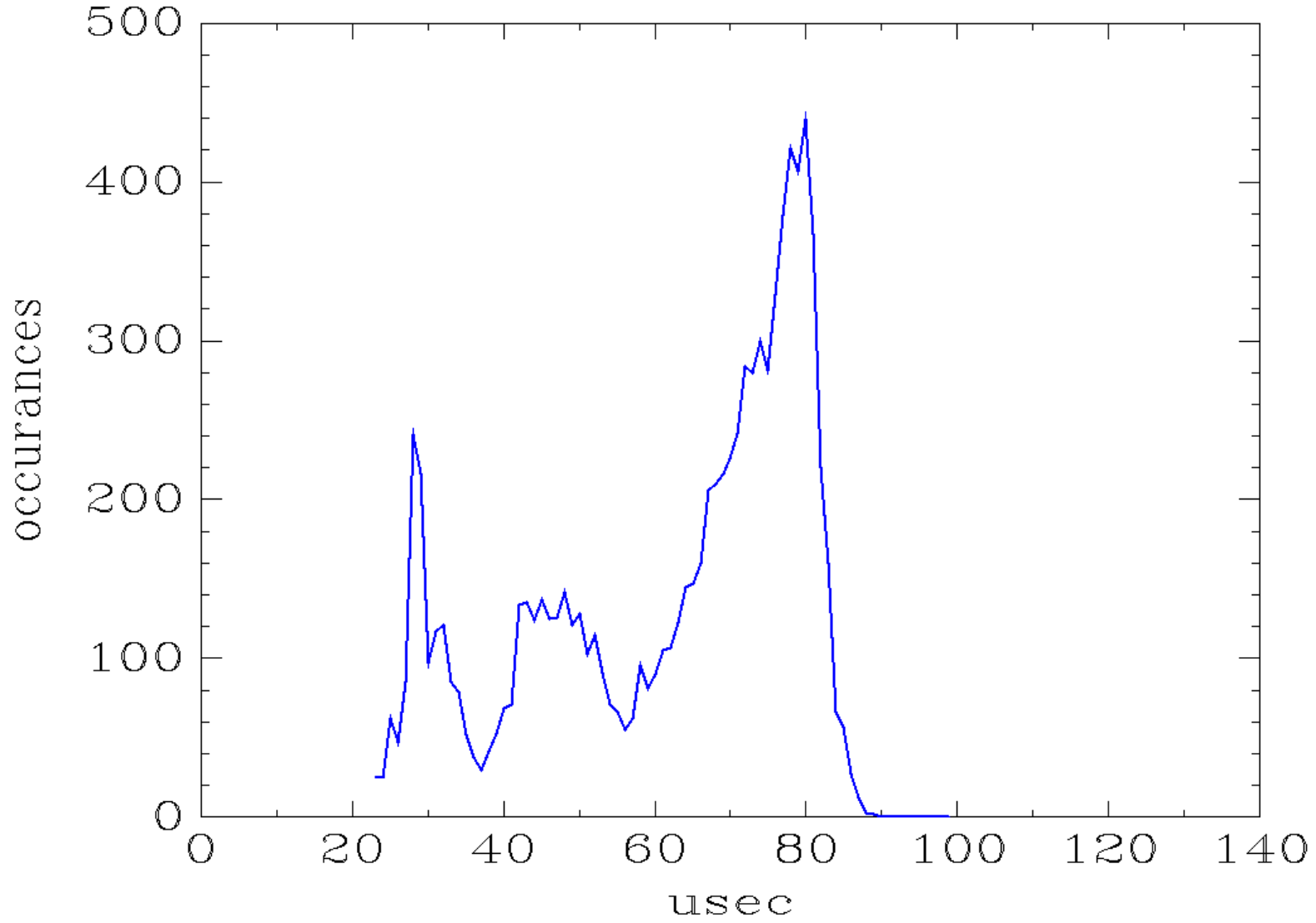
Reality Check - Life Is Not Simple

The next slides show that when the same test is measured many times, there are several distinct patterns that can be seen.

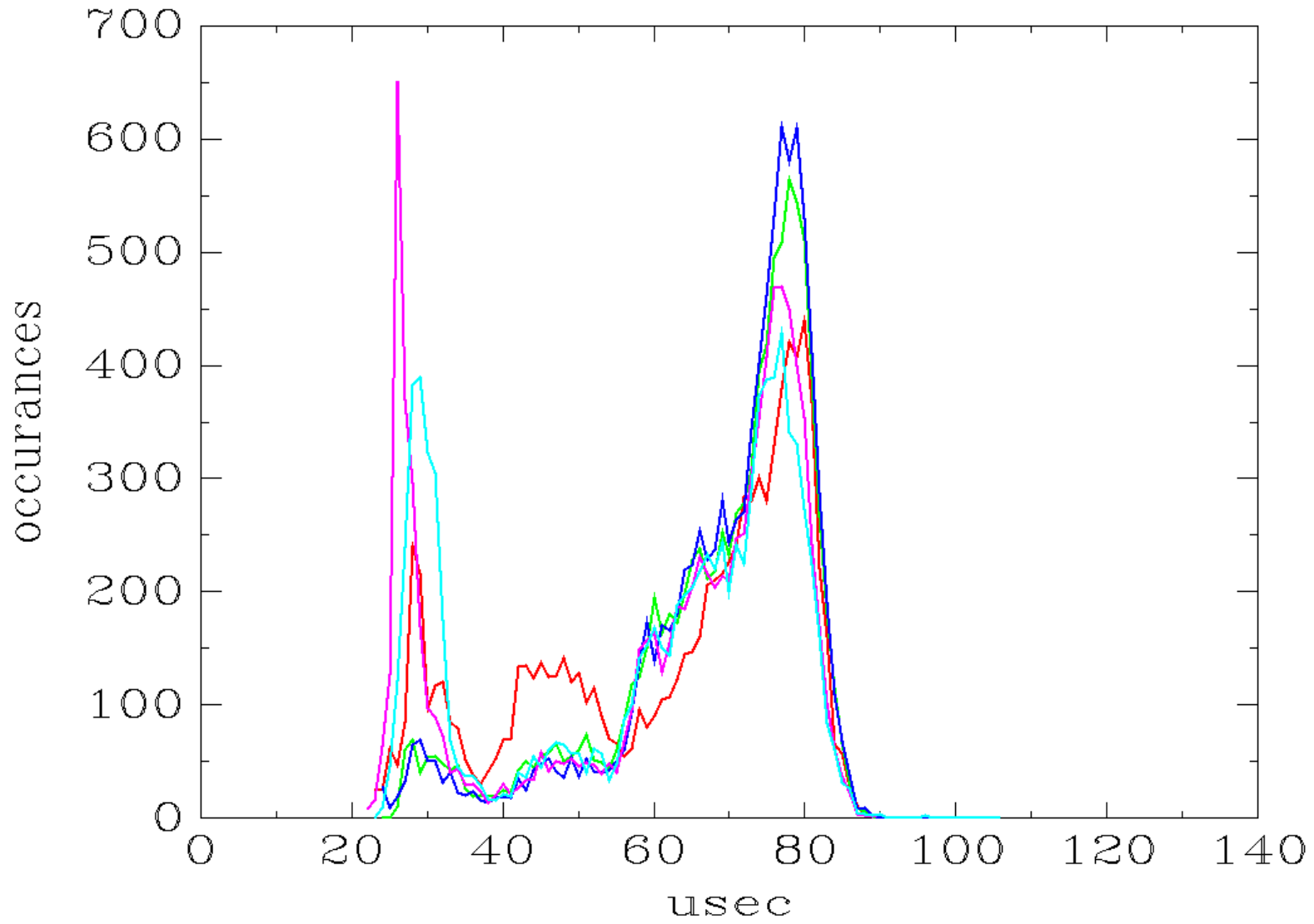
This presentation has ignored such complexity and only presented a single pattern of most graphs.

When you measure your own system, expect to have to analyze the messier picture.

hist_irqs_off_27_h-1_c-0_1
cpu 0



irqs_off_27
cpu 0



Review and Loose Ends

Existing kernel instrumentation sometimes provides imprecise or insufficient data for finding the last 10% of the causes of excessive latency.

This may result in guessing about the causes (aka “informed theories”).

Review and Loose Ends

The data required to confirm or disprove the informed theories can be acquired by extending existing tools or creating new tools.

Improved tools can also provide the insights needed to be able to ask the correct question.

Hardware performance monitor data can be extremely useful.

Review and Loose Ends

This presentation provided some examples of making inferences, creating theories, ruling out incorrect theories, collecting data to confirm correct theories, and gaining new insights from enhanced data collection.

Review and Loose Ends

Conclusions not surprising, but attribution required extended instrumentation and analysis tools.

I-cache issues can have a negative impact on maximum **interrupts disabled duration**.

An L2 cache can decrease maximum **interrupts disabled duration**.

runqueue lock contention can have a negative impact on maximum **interrupts disabled duration**.

How To Collect PMU Data?

Older Kernels

- “Roll Your Own”
- Out of Tree Tools

perfmon2

<http://perfmon2.sourceforge.net/>

How To Collect PMU Data?

Newer Kernels

- “Roll Your Own”

How To Collect PMU Data?

Newer Kernels

- perf

`$KERNEL_SRC/tools/perf`

`$KERNEL_SRC/tools/perf/Documentation/`

https://perf.wiki.kernel.org/index.php/Main_Page

How To Collect PMU Data?

Newer Kernels

- perf

It should be possible to create a PMU trace of `do_local_timer()` using the perf kernel API.

I have not tried this yet.

Questions?

How to get a copy of the slides

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

