Interrupts Considered Harmful

Peter Chubb and Yang Song

first.last@ nicta.com.au

This work arose after reading Luis Henrique's critique of threaded interrupts in RT-Preempt [Hen09].
The work was paid for by NICTA. NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

Quick history of Computing

- All I/O originally programmed
- Desire for multitasking
- . . . Interrupts

When computing was first invented, all I/O was polled — when you wanted to get a bit of data, you'd get the machine to wait until the hardware was ready, then read or write the appropriate registers.
But waiting around for a slow device wastes (expensive) compute time.

INTERRUPTS

- Allow I/O in parallel with computation
  - Good!
- Steal time from main process
  - Bad!

So the hardware designers in the 1950s decided to allow a peripheral processor to *interrupt* the central processor when it wanted service. This allows the central processor to keep doing useful work in parallel with I/O.
However, interrupt servicing steals time from the main task(s). When peripherals are slow relative to the CPU, that doesn't matter — the time spent servicing interrupts is small compared with the time spent on the main task.
But what happens if a peripheral interrupts too often? Or the interrupt service routine (ISR) takes too long?

TRADITIONAL IMPLEMENTATION

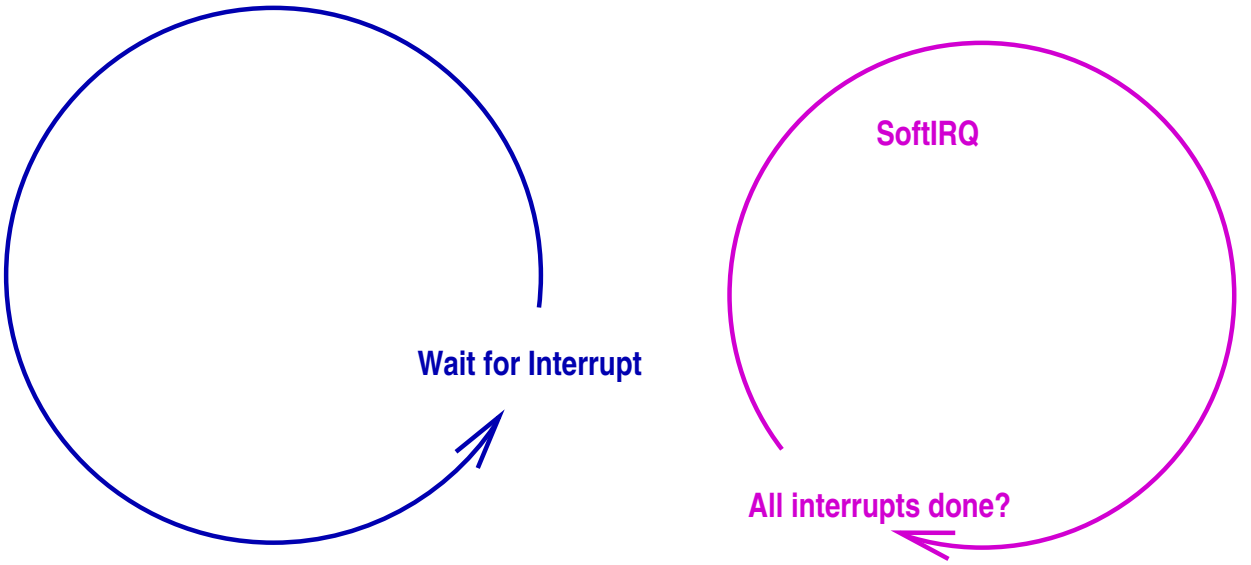
In a traditionally arranged UNIX system, an ISR steals time from and runs on the kernel stack of the process that was running at the time of the interrupt. ISR time is not accounted for separately, so if there are many interrupts, task scheduling will be perturbed; while an ISR is running, no other processing can take place on that processor.

Traditional Implementation

- Interrupts higher priority than other activities.
- Interrupts *prioritised* — higher priority interrupts can interrupt lower ones.
- Real-time tasks are preempted by interrupt servicing.
- Long running ISRs *defer work* to, e.g., a *soft IRQ* handler.

Some architectures allow interrupts to be nested, but as they share the same (small) kernel stack they may not be nested too deeply.
If an ISR needs to run for more than a very short time, it will defer some of its work, so that other interrupts can be serviced in a timely manner. The driver we'll analyse (for the e1000) defers work to a *softirq*, which is serviced like any other ISR but at a lower priority than (and therefore interruptible by) any real hardware interrupt line.
Interrupt handling is the highest priority activity in a traditional system — interfering with real-time tasks, and possibly causing them to miss their deadlines.

THREADED INTERRUPTS



The way threaded interrupts work is that each interrupt is given a thread. That thread waits for an interrupt; when the interrupt happens it continues, services the device, then when it has done all its work, waits for the next interrupt. Using this approach, a driver can be structured as a state machine, with interrupts being just one of the events that causes transitions.
Linux puts the softirq handler into a thread as well, prioritised lower than hardware interrupt threads. This allows threaded interrupts with minimum change to the existing (legacy) driver structure.

Threaded Interrupts

- Threads can be prioritised against real-time work
- Threads can sleep — no longer any need for deferred work
  - Except in special cases
- Easier to program.

One alternative is to put each ISR into a kernel thread. The way this works is to split the ISR into two parts: a very small and simple stub merely disables the source of the interrupt (either by talking to the interrupt controller chip (APIC), or by being part of the driver and talking to the interrupting hardware), then makes the ISR thread runnable.
A number of operating systems do this, including Solaris 2 [KE95], FreeBSD, and Tunis [Hol83]. The advantages are that because interrupts are controlled individually (and so need not block lower priority ones) there is no longer any need to defer work; the ISR can sleep or be preempted (easing the programmer's burden), and as kernel threads have priorities, the ISR thread can be prioritised against any real-time tasks.

THREADED INTERRUPTS

- Linux *can* use threaded ISRs
  - Doesn't for e1000.
- RT-PREEMPT patch forces threaded ISRs everywhere.

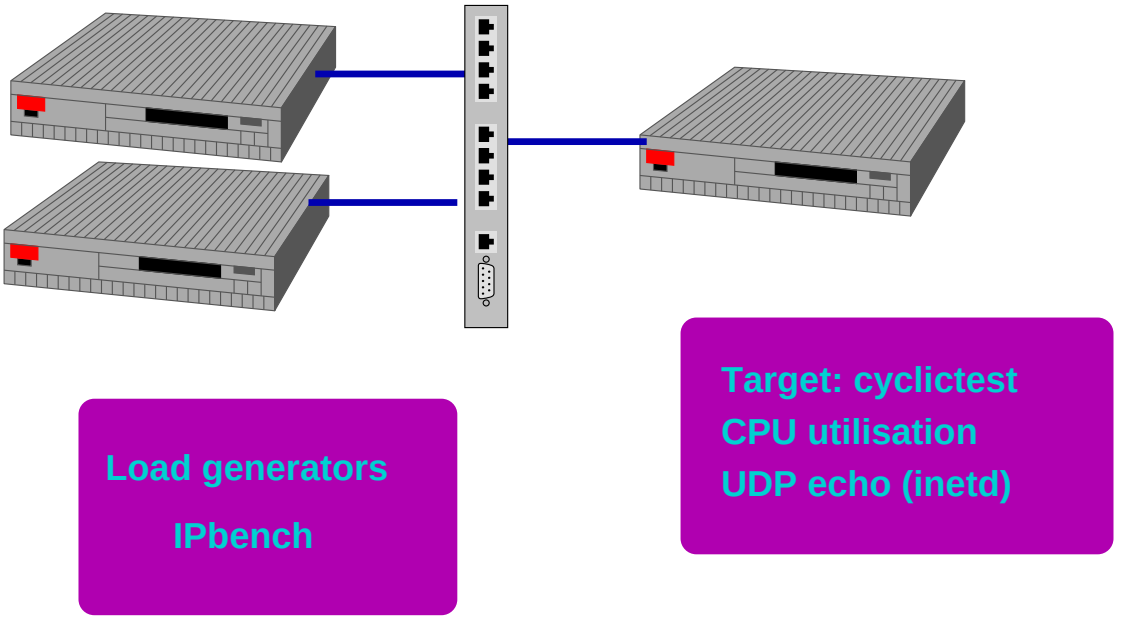
There are two ways to use threaded interrupts in Linux. Recent Linux kernels (since about 2.6.30) have a `request_threaded_irq()` call. Also the softirq handlers run in a separate thread now. However, only a few drivers use threaded interrupts at present.
Alternatively, an out-of-tree patch RT-PREEMPT [Pre09] makes all interrupts to be handled as threads (and therefore preemptible).

RT-PREEMPT PATCH

- Adds many preemption points to kernel
  - Aims to be *fully preemptive*
- Makes *all* interrupts have threaded ISRs
  - So they can be preempted.
  - So they can be prioritised against real-time tasks.

The main purpose of the RT-PREEMPT patch is to reduce latency for real time tasks. It does this by introducing new preemption points, including in interrupt handlers — and if ISRs are preemptible then *all* interrupt handlers must be threads.

TEST BENCH



Luis Henriques of Intel [Hen09] evaluated performance for RT-PREEMPT's threaded interrupts, and found it poor. We decided to try and find out why: is this just because of using threaded interrupts, or a feature of the implementation?

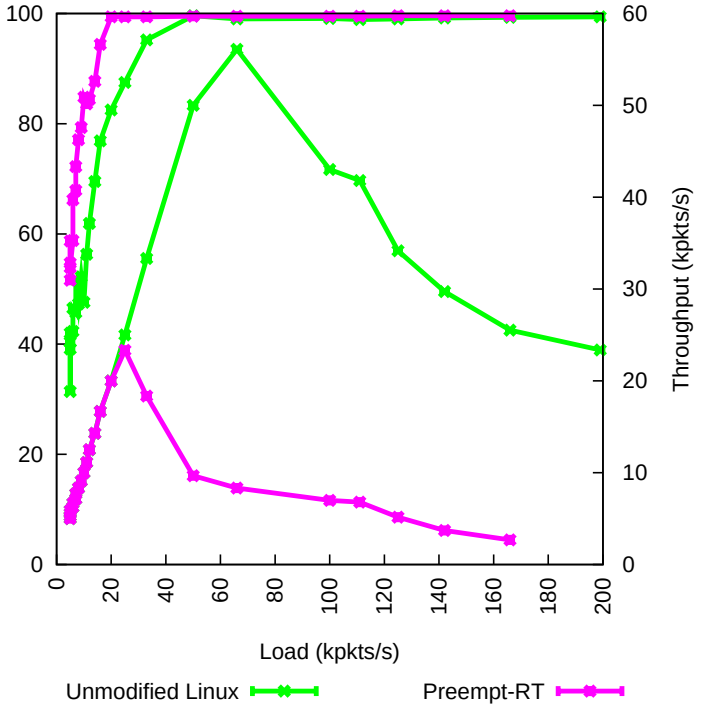
We set up a test system with a few load generators and a target connected by a low-latency Gigabit switch. The systems were all identical Celeron 2GHz machines with e1000 Gigabit ethernet adapters, running Debian, with custom compiled 2.6.31.4 kernels.

We used the *ipbench* [WM04] to generate an interrupt load. Ipbench is a configurable network performance suite that can measure throughput and individual packet latency while running a co-benchmark on the target. In this case, **`inetd`** on the target was configured to echo UDP packets; the load generators sent 64-byte UDP packets to the target at a set rate, and timed each one until it returned.

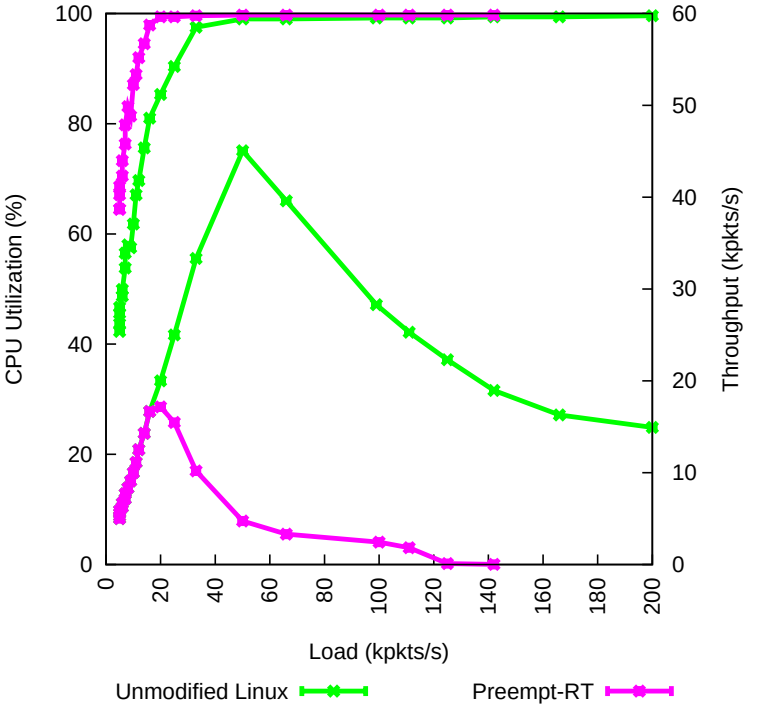
A CPU usage meter and *cyclictest* [Gle09] were run as co-benchmarks on the target. *Cyclictest* is a program that sleeps for a set period of time, and then measures the time from when it was supposed to wake up to when it is actually scheduled — it thus acts as a crude real-time latency test.

We configured, *cyclictest* to sleep for 500μs at a time, and to run repeatedly for 100 000 loops to collect the average and maximum latency time. *Cyclictest* ran in the **`SCHED_FIFO`** real-time scheduling class at a priority higher than any other threads in the system including interrupts.

BASELINE PERFORMANCE



Without cyclictest

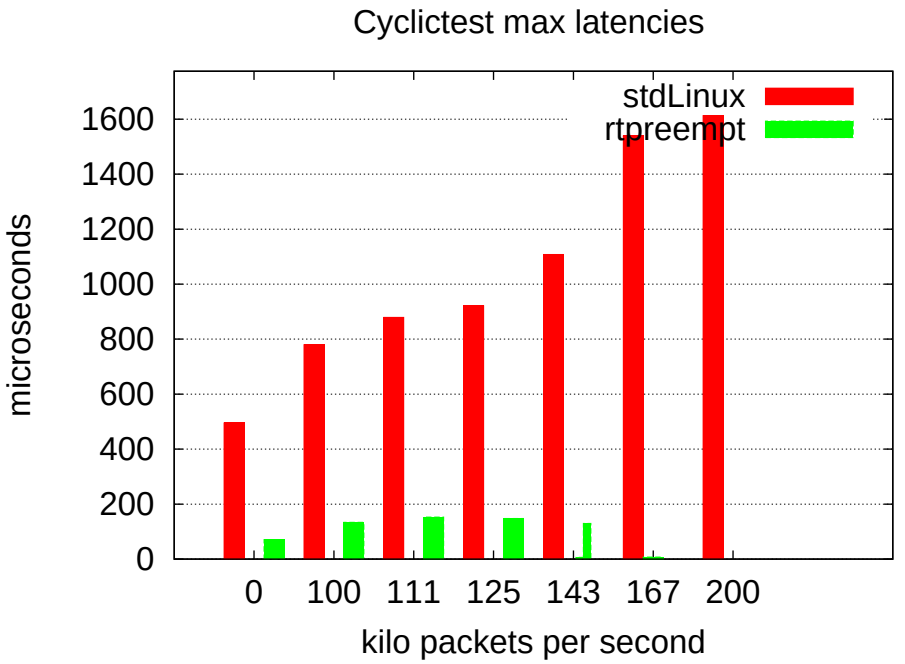


With cyclictest

You can see from the graphs that when the packet rate starts to increase, performance sags. RT-PREEMPT performance is very poor; at a 5μs inter-arrival time, no packets are echoed.

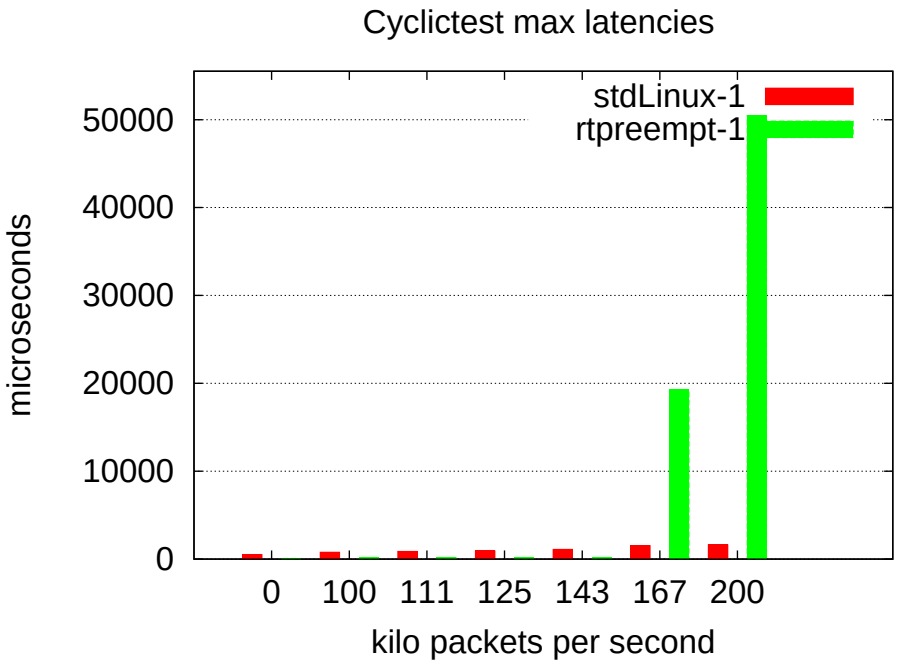
With the *much* poorer performance of RT-PREEMPT we'd expect more time available for the real-time process

BASELINE PERFORMANCE



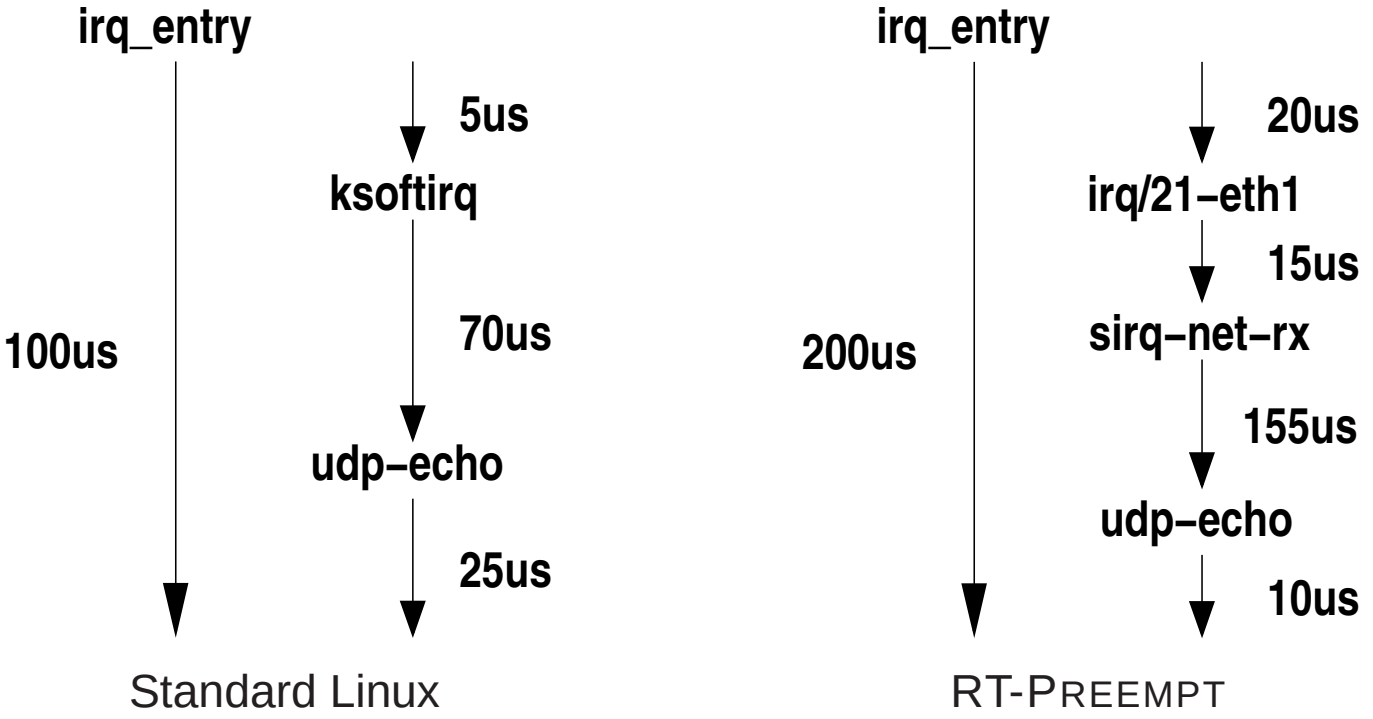
Down to 7μs between packets (143kpps) this seems to be so: cyclictest reports latencies below 200μs.

BASELINE PERFORMANCE



But at higher packet rates latency soars to over 20ms.

WHERE DOES THE TIME GO?



We ran some traces (using *ftrace*) on an overloaded system. It's apparent that the (non-preemptible) softirq is stealing time from *echo* (giving poor performance), and also delaying cyclictest.

The softIRQ here processes all the packets queued from the card, allocating memory for skbuffs, pushing the packets up the TCP/IP stack, and waking user-space if necessary.

Also, if the softirq is running when the timer interrupt fires, instead of *cyclictest* running next, the softirq is continued. This is why there are 20ms latencies — the softirq for the 5μs load can take a *very* long time.

WHERE TO FROM HERE?

- Not enough time in UDP Echo
- Too much time in softIRQ?
- Work on throughput first.
- *then* check latencies

We decided to attempt to fix the throughput issue first, then revisit the latencies.

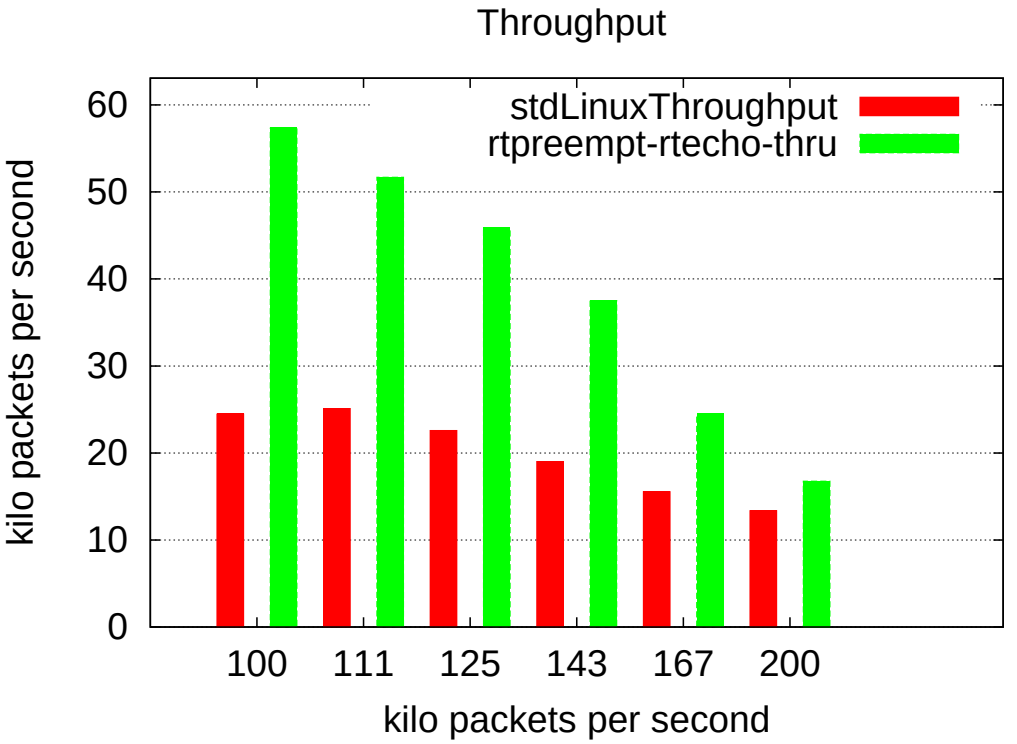
WHERE TO FROM HERE?

Step one

Make *echo* real-time, same priority as *softirq*.

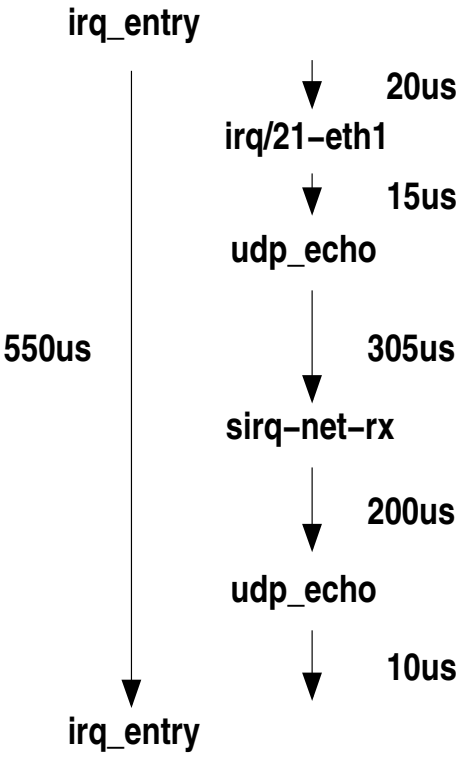
So we tried to make *echo* a real time thread, with the same priority as the softirq. This should cause better sharing between them. (We also tried to make it a higher priority, but that caused too little work to be done in the softIRQ).

REAL-TIME USER-SPACE



The throughput looks a lot better now — even better than the non-threaded case we had at first.

REAL-TIME USER-SPACE



The softIRQ time and user-space time are better balanced now, leading to better throughput — but too much time is still being spent in the softirq relative to user-space udp-echo!

REAL-TIME USER-SPACE

- Still too much time in softIRQ
  - ➜ packets queued for later dropping
  - ➜ (wasted time)
- restrict work in softirq

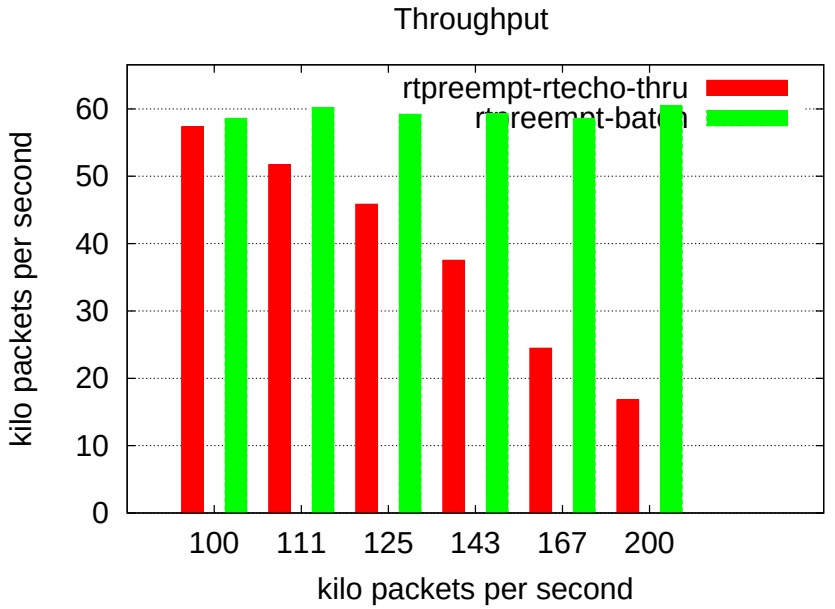
Any work done on packets that will later be dropped is totally useless. So (following Macpherson [Mac07]) we'll attempt to reduce the time spend in the soft IRQ. This should fix some of the *cyclictest* latency, too.

RESTRICT WORK IN SOFTIRQ

- cap batch at 32 packets
  ➜ (or 16 or 64 ...)
- regular preemptions allow *echo* to make forward progress

So we altered the softirq routine in the e1000 driver to process a batch of at most 32 packets before allowing a preemption. This allows *echo* to make forward progress. (We also tried 64 packet batches with slightly better throughput).

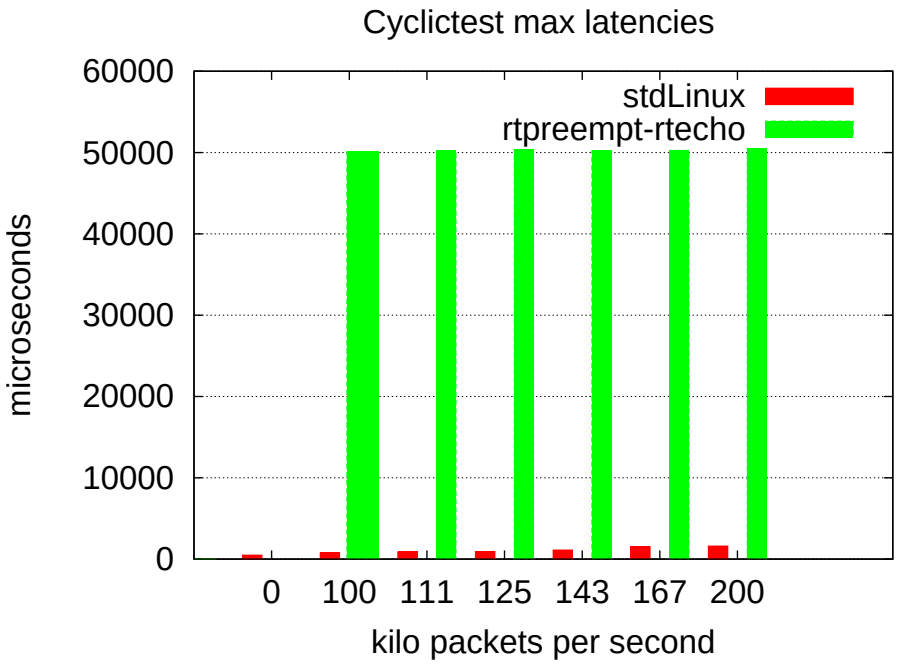
RESTRICT WORK IN SOFTIRQ



- Much better throughput
- But what of cyclictest latency?

As can be seen, the throughput is much improved. But let's have a look at the cyclictest latency...

RESTRICT WORK IN SOFTIRQ



It's *appalling*! 50ms latency!

Ooops

Oops indeed.

RESTRICT WORK IN SOFTIRQ

The machine stalls for 50ms every second

run-balancing in the scheduler

Can't run a system 100% Real-Time!

It turns out that to prevent people shooting themselves in the feet, the Linux scheduler will not *allow* a 100% real-time load. After each 0.95s it checks to see if any non-real-tine thread has been scheduled, and if not, it schedules one (or the idle thread). This theoretically is to allow an administrator to login in and kill a runaway real-time process.

HERESY?

- Try running IRQ thread in SCHED_NORMAL
- Adjust *nice* to get good balance

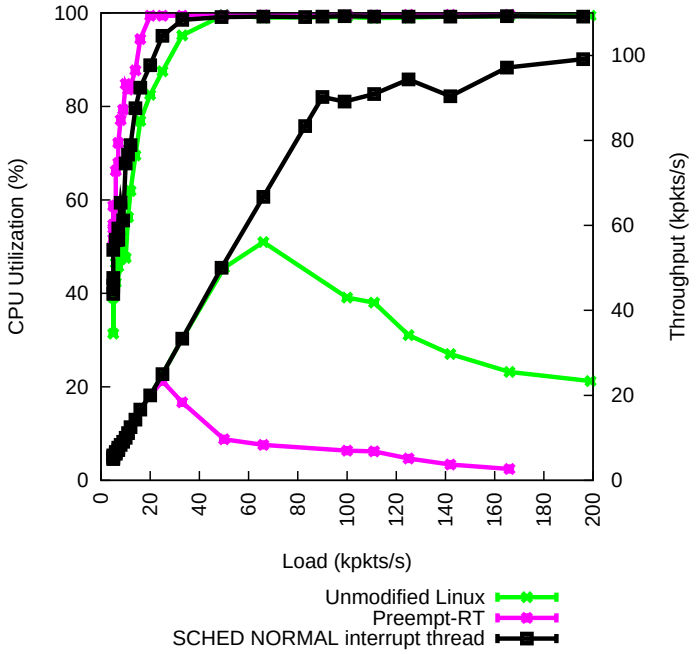
OK, let's try running the interrupt thread as a time-share process. In Linux, decreasing one nice unit means 10% more time. We traced the system and found that the user-space echo process needs around 1.5 times the time that the softIRQ does per packet, so we'll try nice 4 and 5.

HERESY?

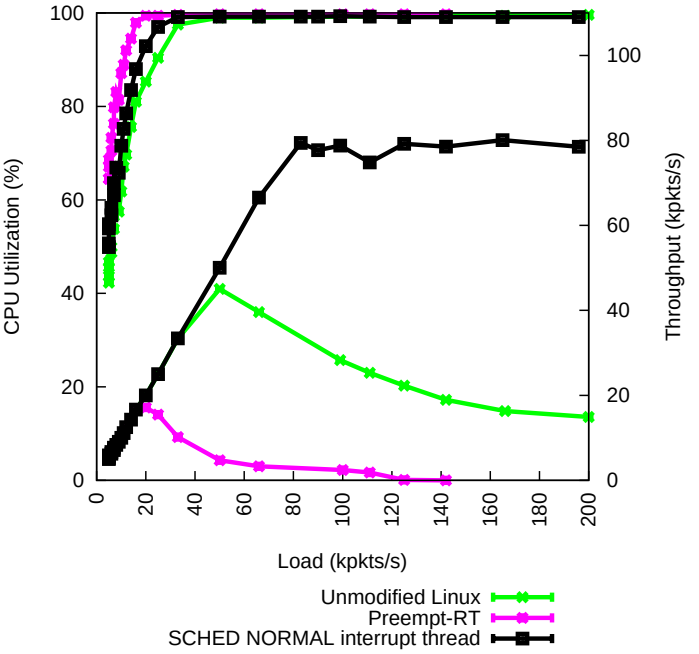
- Standard Linux kernel 2.6.31.4
- Change e1000 driver to:
  - ➜ request threaded interrupt handler
  - ➜ set thread to `SCHED_NORMAL`
  - ➜ no deferred work — no separate softIRQ
  - ➜ *nice* thread
  - ➜ *batch* packets — max 32 at a time before `preempt()` call
- *cyclictest* only real-time task

As cyclictest is now the *only* real time task, run balancing in the scheduled won't hurt us. We'll combine the deferred work into the interrupt thread, and run it at *nice* 5, giving a 1.6:1 time ratio between user-space at the default nice, and the softIRQ thread. Obviously this only works for this application.

HERESY — BUT IT WORKS!



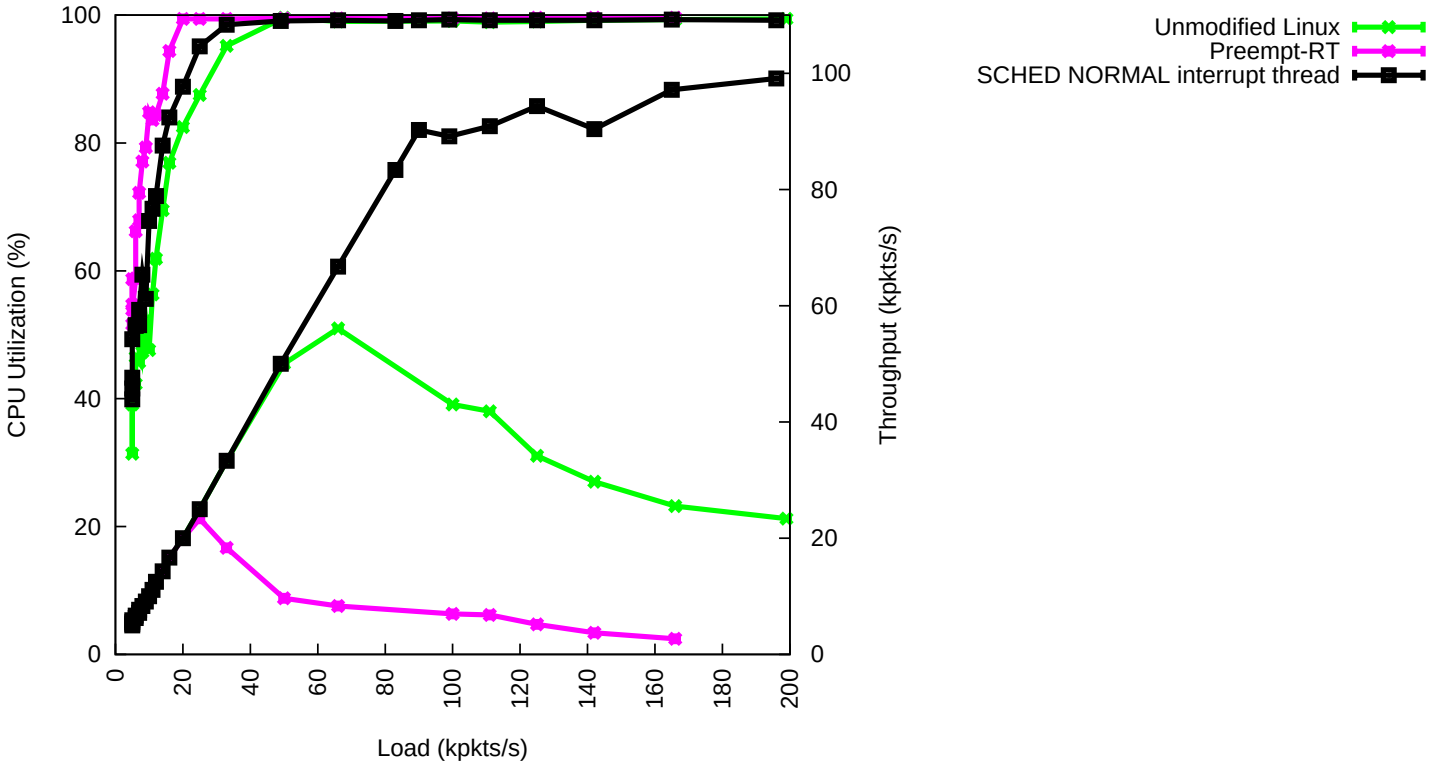
Without cyclictest



With cyclictest

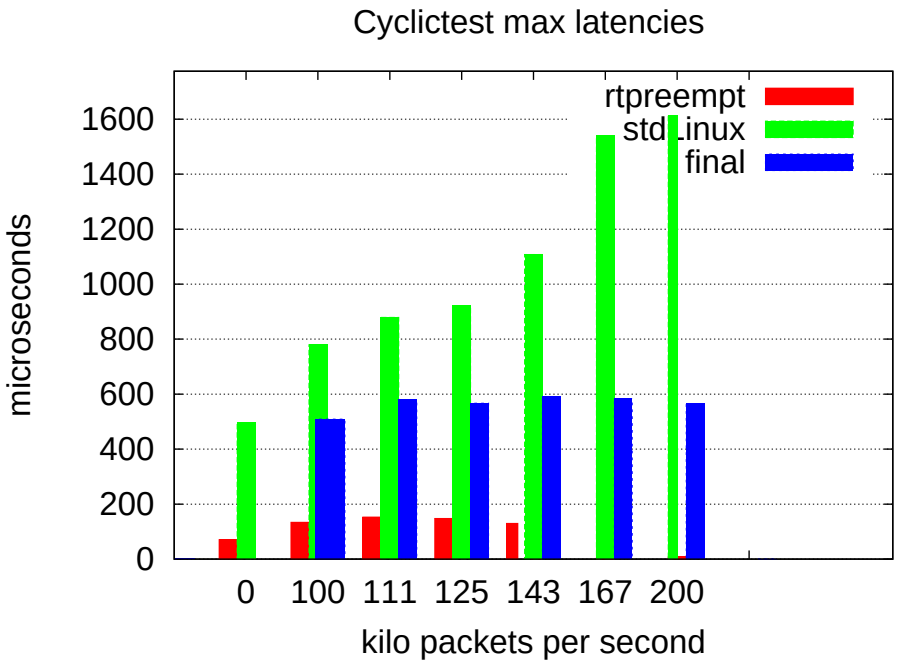
Ooooooohhhh —- performance is really good. We're outperforming all of the previous systems by a large amount.

HERESY — BUT IT WORKS!



Let's have a better look (without cyclictest running). The performance peaks at around 100kpps, and doesn't drop under increased load.

HERESY — BUT IT WORKS!



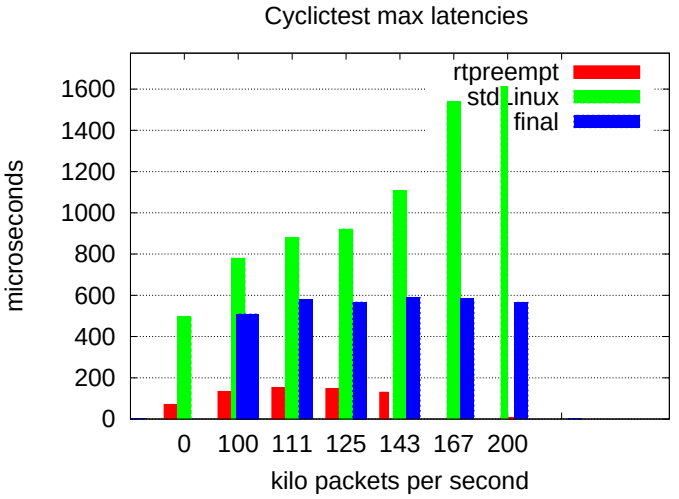
And cyclictest latencies are reasonable. They about three times the low packet rate RT-PREEMPT latencies, but do not climb massively under high packet loads. And investigating and adding additional preemption points will probably reduce them.

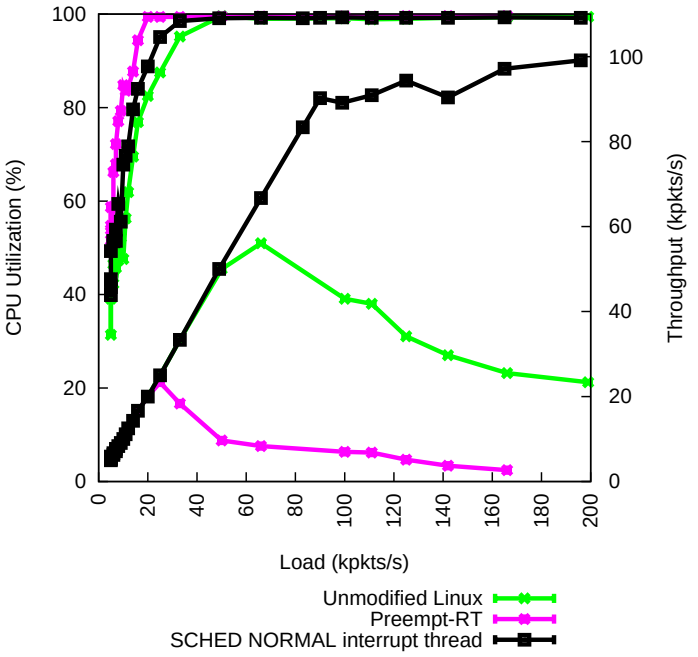
Future Work

- automatic *nice* (or handshaking)
- Test with other loads (e.g., HTTP, NFS, scp)
- Test other drivers.
- Investigate remaining latencies.
- Clean up patches and submit.

So, obviously, the solution we came up with works only for this workload. We need to provide a mechanism for changing the amount of time given to the interrupt thread automatically, using some kind of feedback from higher levels perhaps.
Also we need to look at other systems like this. Network is essentially a producer-consumer situation, but with no handshaking between producer and consumer. The way we're controlling the relative times may be better done as an explicit handshake.

FINAL OUTCOME





References

[Gle09] Thomas Gleixner. cyclictest. http://rt.wiki.kernel.org/index.php/ Cyclictest/, June 2009.

[Hen09] Luis Henriques. Threaded IRQs on Linux PREEMPT-RT. In *International Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, pages 23–32, Dublin, Ireland, June 2009. http://www.artist-embedded.org/artist/Overview,1750.html.

[Hol83] R. C. Holt. *Concurrent Euclid, The UNIX system, and TUNIS*. Addison-Wesley, 1983.

[KE95] Steve Kleiman and Joe Eykholt. Interrupts as threads. *ACM Operating Systems Review*, 29(2):21–26, 1995.

[Mac07] Luke Macpherson. *Performing Under Overload*. PhD thesis, School of Computer Science and Engineering, University of NSW, Sydney 2052, Australia, September 2007. Available from publications page at `http://www.disy.cse.unsw.edu.au/`.

[Pre09] PREEMPT-RT website. http://rt.wiki.kernel.org/index.php/ Main_Page, 2009.

[WM04] Ian Wienand and Luke Macpherson. ipbench: A framework for distributed network benchmarking. In *AUUG Winter Conference*, Melbourne, Australia, September 2004.