

Linux kernel preemption and the latency-throughput tradeoff

Dec 23, 2019

What is preemption?

Preemption, otherwise known as preemptive scheduling, is an operating system concept that allows running tasks to be forcibly interrupted by the kernel so that other tasks can run. Preemption is essential for fairly scheduling tasks and guaranteeing that progress is made because it prevents tasks from hogging the CPU either unwittingly or intentionally. And because it's handled by the kernel, it means that tasks don't have to worry about voluntarily giving up the CPU.

It can be useful to think of preemption as a way to reduce scheduler latency. But reducing latency usually also affects throughput, so there's a balance that needs to be maintained between getting a lot of work done (high throughput) and scheduling tasks as soon as they're ready to run (low latency).

The Linux kernel supports multiple preemption models so that you can tune the preemption behaviour for your workload.

The three Linux kernel preemption models

Originally there were only two preemption options for the kernel: running with preemption on or off. That setting was controlled by the kernel config option, `CONFIG_PREEMPT`. If you were running Linux on a desktop you were supposed to enable preemption to improve interactivity so that when you moved your mouse the cursor on the screen would respond almost immediately. If you were running Linux on a server you ran with `CONFIG_PREEMPT=n` to maximise throughput.

Then in 2005, Ingo Molnar introduced a third option named `CONFIG_PREEMPT_VOLUNTARY` that was designed to offer a middle point on the latency-throughput spectrum – more responsive than disabling preemption and offering better throughput than running with full preemption enabled. Nowadays, `CONFIG_PREEMPT_VOLUNTARY` is the default setting for pretty much all Linux distributions since [openSUSE switched](#) at the beginning of this year.

Unfortunately, choosing the best Linux kernel preemption model is not straightforward. Like with most performance topics, the best way to pick the right option is to run some tests and use cold hard numbers to make your decision.

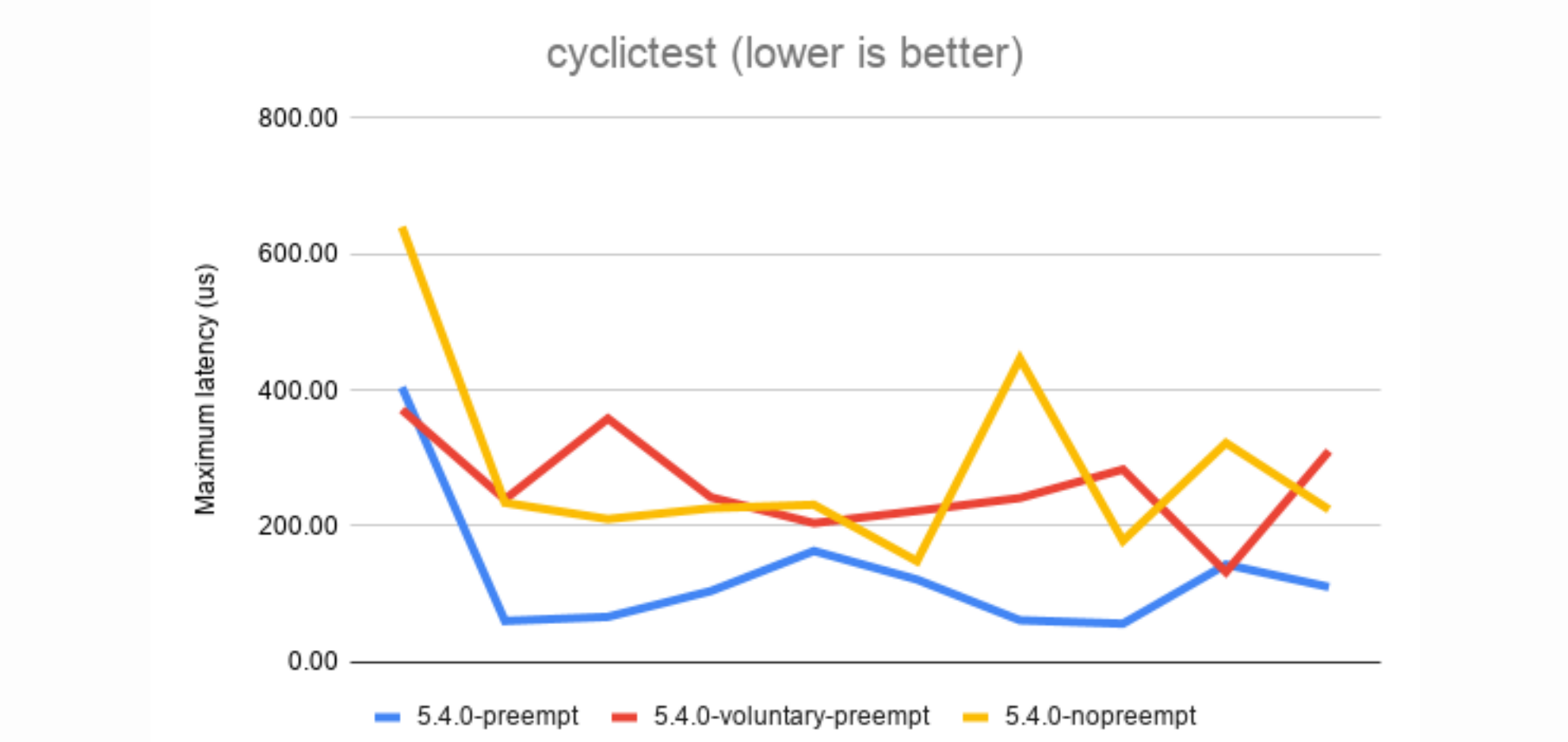
What are the differences in practice?

To get an idea of how much the three config options lived up to their intended goals, I decided to try each of them out by running the [cyclicttest](#) and [sockperf](#) benchmarks with a Linux 5.4 kernel.

If you're interested in reproducing the tests on your own hardware, here's how to do it.

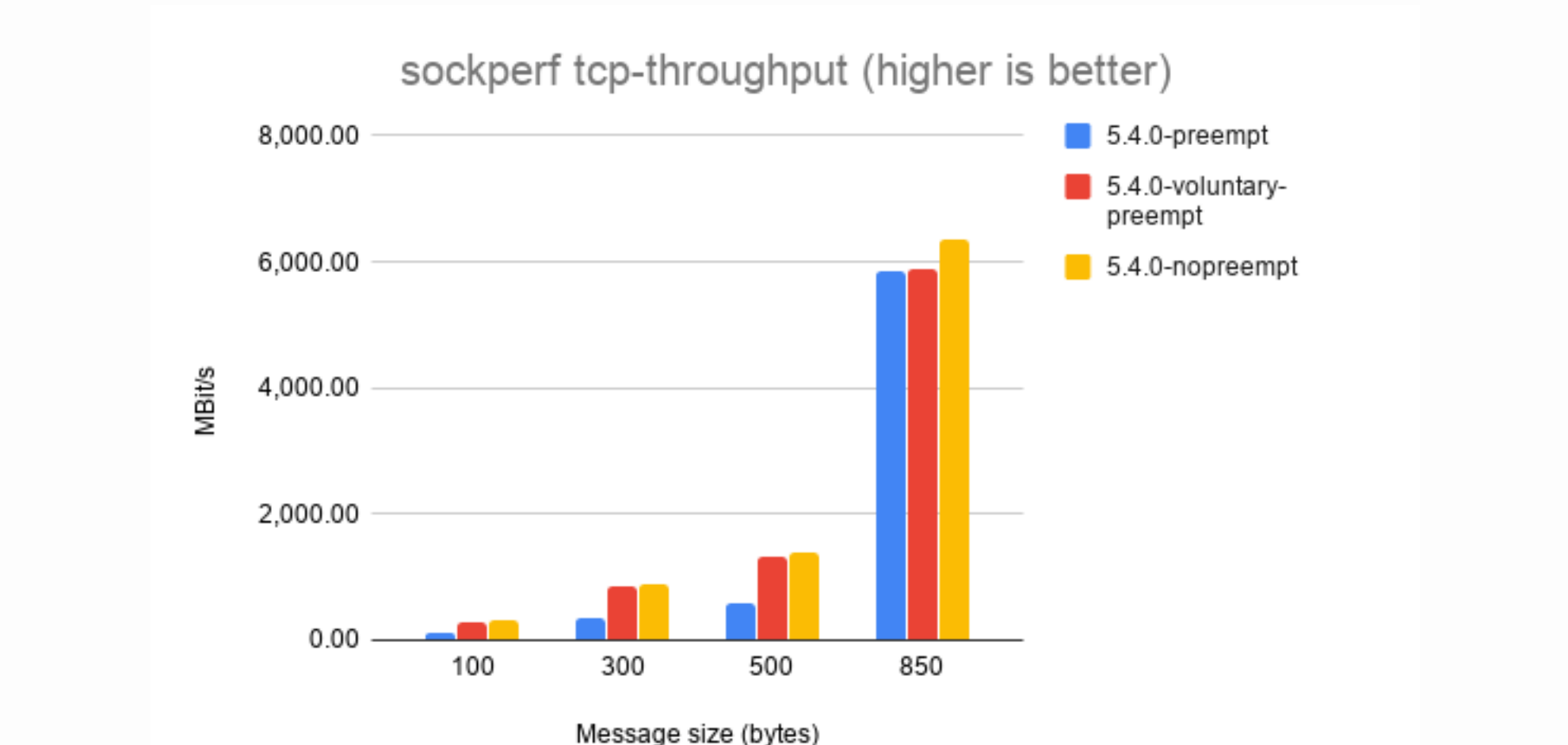
```
$ git clone https://github.com/gormanm/mmtests.git
$ cd mmtests
$ ./run-mmtests.sh --config configs/config-workload-cyclicttest-hackbench `uname -r`
$ mv work work.cyclicttest && cd work.cyclicttest/log && ../../compare-kernels.sh | less
$ cd ..
$ ./run-mmtests.sh --config configs/config-network-sockperf-pinned `uname -r`
$ mv work work.sockperf && cd work.sockperf/log && ../../compare-kernels.sh | less
```

cyclicttest records the maximum latency between when a timer expires and the thread that set the timer runs. It's a fair indication of worst-case scheduler latency.



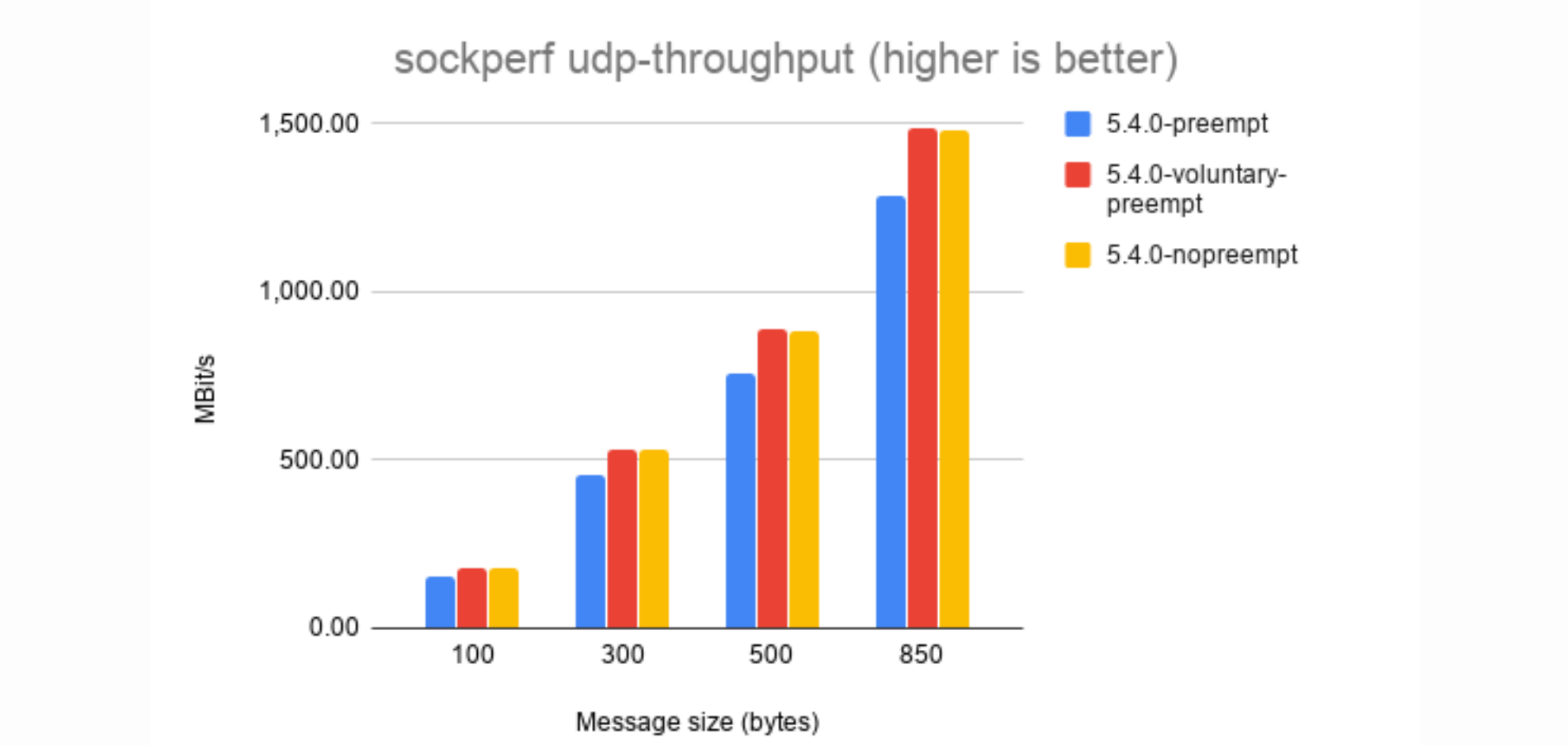
The above results show that the best (lowest) latency is achieved when running with `CONFIG_PREEMPT`. It's not a universal win, as you can see from the first data point. But overall, `CONFIG_PREEMPT` does a decent job of keeping those latencies down. `CONFIG_PREEMPT_VOLUNTARY` is a good middle ground and exhibits slightly worse latency while `CONFIG_PREEMPT_NONE` shows the the worst (highest) latencies of all. Based on the descriptions of the kernel config options given in the preemption models section, I'm sure we can all agree these are roughly the results we expected to see.

Next, let's look at sockperf's TCP throughput results. sockperf is a network benchmark that measures throughput and latency over TCP and UDP. For this experiment, we're only interested in the throughput scores.



It's a little hard to make out some of the results, but each of the different message sizes shows that `CONFIG_PREEMPT_NONE` achieves the best throughput, followed by `CONFIG_PREEMPT_VOLUNTARY` and with `CONFIG_PREEMPT` coming last. Again, this is the expected result.

Things get a little weirder with sockperf's UDP throughput results.



Here, `CONFIG_PREEMPT_VOLUNTARY` consistently achieves the highest throughput. I haven't dug into exactly why this might be the case, but my best guess is that preemption doesn't matter as much for UDP workloads because it's stateless and doesn't exchange multiple messages between sender and receiver like TCP does.

If you've got any ideas to explain the UDP throughput results please leave them in the comments!

Tweet

Share

ALSO ON CODE BLUEPRINT BLOG

A Kernel Dev's Approach to ...

5 years ago • 3 comments

In the last post I talked about tuning mutt to reduce the time to open my ...

When Was The Last Time You Were In ...

5 years ago • 3 comments

By pure coincidence, two of the books I've recently read discussed the topic ...

The Linux x86 ORC Stack Unwinder

4 years ago • 11 comments

No one wants their Linux machine to crash. But when it does, providing as ...

Dear Btrfs, fsck is my f

5 years ago • 3

After booting r Tumbleweed li morning, I was

8 Comments Code Blueprint Blog Disqus' Privacy Policy Login

Favorite Tweet Share Sort by Best

Join the discussion...

LOG IN WITH OR SIGN UP WITH DISQUS

joelagnel · 2 years ago

Oh, and also PREEMPT_RT comparison might also be cool to add to the mix.

1 ^ | · Reply · Share

Matt Fleming · Edited · joelagnel · 2 years ago

Definitely. I'll get PREEMPT_RT numbers.

1 ^ | · Reply · Share

joelagnel · 2 years ago

My guess is voluntary preemption points might just be yielding control at the right time adding just the right amount of context switch overhead. Where as involuntary preemption just waits for time slice to expire and may be causing too many preemptions for this workload. Could you also count how many context switches you have in various test configurations?

1 ^ | · Reply · Share

Anderson Silva · 2 years ago

Wow very interesting article, thank you sir

1 ^ | · Reply · Share

James M · 2 years ago · edited

There's a lot of confusion in this area. Can you do an article on preempt vs "realtime" vs "tickless" vs CONFIG_HZ adjustments. Even within CONFIG_HZ adjustments people are still claiming vastly different optimizations for the same things ("100hz/250hz/1000hz is best"). It seems realtime kernel is recommended for audio applications often, while the rest of the above are not... I think a spreadsheet of patch-name on x-axis && pros/cons on y-axis would be very useful!

^ | · Reply · Share

Kernel dev · 2 years ago · edited

I guess the reason the results are "strange" is that scheduler latency may not be a factor in the measurement. You could make it a factor by generating stress in the scheduler by starting a number of CPU bound threads for example.

^ | · Reply · Share

Neil Gunther · 2 years ago

The reason the packet thruput results seem to run counter to expectations, might have to do with the CPU service times being size-dependent. Preempting computation for a variety of network payloads most likely conflicts with what the packetizer thinks it should be doing. It might even cause it start over, in certain circumstances.

In my experience, preemption control is usually best directed at higher level application performance (e.g., database transactions).

^ | · Reply · Share

Neil Gunther · 2 years ago

It's hard to draw conclusions when the workloads for latency (R) and thruput (X) differ. I would suggest measuring both X and R for the same workload and also make the workload a little more systems level (e.g., DB txs), if possible.

^ | · Reply · Share

Subscribe Add Disqus to your site Do Not Sell My Data

DISQUS