

The Humble Bridge

BY KRISTOF PROVOST

The humble bridge. Connecting places across rivers, ravines, chasms, valleys, ...

No, wait, we're not talking about that sort of bridge. `if_bridge` links together two or more Ethernet links (IEEE 802 networks, if you're that kind of person), in effect, turning your computer into a switch without all the messy cables.

One of the common use cases for `if_bridge` is to connect multiple virtual machines (or VNET jails) to a real network interface.

For example, this particular machine has a few bhyve VMs on it:

```
vmbridge: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> metric 0 mtu 1500
ether 02:ec:9a:db:86:01
inet 172.16.1.1 netmask 0xfffff00 broadcast 172.16.1.255
id 00:00:00:00:00:00 priority 32768 hellotime 2 fwddelay 15
maxage 20 holdcnt 6 proto rstp maxaddr 2000 timeout 1200
root id 00:00:00:00:00:00 priority 32768 ifcost 0 port 0
member: tap5 flags=143<LEARNING,DISCOVER,AUTOEDGE,AUTOPTP>
ifmaxaddr 0 port 11 priority 128 path cost 2000000
member: tap4 flags=143<LEARNING,DISCOVER,AUTOEDGE,AUTOPTP>
ifmaxaddr 0 port 10 priority 128 path cost 2000000
member: tap2 flags=143<LEARNING,DISCOVER,AUTOEDGE,AUTOPTP>
ifmaxaddr 0 port 7 priority 128 path cost 2000000
groups: bridge
nd6 options=9<PERFORMNUD,IFDISABLED>
```

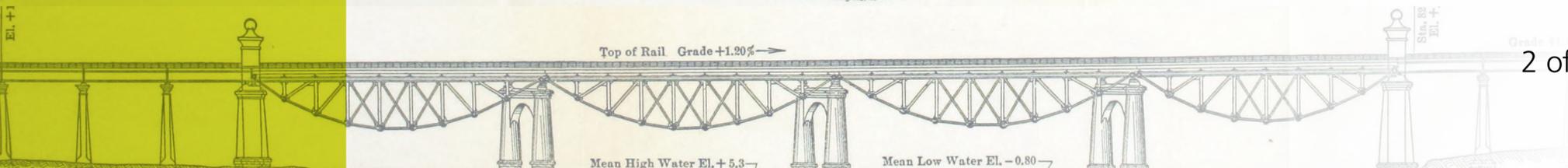
These VMs are connected to the `tap2`, `tap4` and `tap5` interfaces. They can all talk to each other on that particular (virtual) LAN, or to the host via the `172.16.1.1` address. The host machine also helpfully runs a DHCP server and a NAT firewall to translate traffic from the VMs to the public interface, allowing them to access the internet.

Problems

There's nothing particularly wrong with this setup, but if we didn't find a problem, this would be a very short and boring article.

Happily, there is a problem, or there can be a problem for some users. It's not visible, but `if_bridge` does not scale as well as it could. In fact, it currently scales very badly, indeed.

To demonstrate the problem, we'll create a simple test setup consisting of two machines connected with two Chelsio T62100 40Gbps links. The first machine will generate traffic on its



first interface and expect to receive it on the second. The second machine will use `if_bridge` to link its two interfaces.

The Chelseo driver supports `netmap(4)`, which means we can use `pkt-gen` (`pkt-gen-g2017.08.06_1`) to generate traffic. `Netmap(4)` is fast. I'm not going to prove that--you're just going to have to trust me. Yes, really. If you're going to doubt everything I say, we're never going to get anywhere!

So, with the following two commands (the first to receive traffic, the second to generate it) we can run a simple performance test:

```
./pkt-gen -i vcc1 -f rx -w 4 -W
./pkt-gen -f tx -i vcc0 -l 60 -d 192.19.10.1:2000-192.19.10.10 -s 10.10.10.1:2000-10.10.19.143 -S 01:02:03:04:05:06 -D 06:05:04:03:02:01 -w 4
```

We generate 60-byte packets. There's a good reason for this, and it's not just to be difficult.

Well, okay, actually it is precisely to be difficult. In general, it's easier for machines to transmit fewer large packets than many small packets. That's easy to understand, because each packet has a certain amount of overhead (e.g. context switches, working out where to send it, the all-important inter-packet nap, ...). So larger packets are more efficient. In your real network you want large packets, but when we're testing things, we're interested in the worst-case behavior because that's where we'll see the most impact of our attempts to improve things.

Once we run the test, we'll see output like this:

```
957.106303 main_thread [2666] 27.015 Mpps (28.730 Mpkts 12.967 Gbps in 1063499 usec) 497.42 avg_batch 99999 min_space
957.306302 main_thread [2666] 3.668 Mpps (3.901 Mpkts 1.761 Gbps in 1063498 usec) 24.62 avg_batch 949 min_space
958.169302 main_thread [2666] 27.016 Mpps (28.718 Mpkts 12.967 Gbps in 1062999 usec) 497.42 avg_batch 99999 min_space
958.369303 main_thread [2666] 3.669 Mpps (3.900 Mpkts 1.761 Gbps in 1063001 usec) 24.68 avg_batch 885 min_space
959.232304 main_thread [2666] 27.015 Mpps (28.717 Mpkts 12.967 Gbps in 1063002 usec) 497.37 avg_batch 99999 min_space
959.377805 main_thread [2666] 3.669 Mpps (3.700 Mpkts 1.761 Gbps in 1008502 usec) 24.66 avg_batch 950 min_space
960.280311 main_thread [2666] 27.011 Mpps (28.308 Mpkts 12.965 Gbps in 1048007 usec) 497.44 avg_batch 99999 min_space
960.441302 main_thread [2666] 3.668 Mpps (3.901 Mpkts 1.761 Gbps in 1063496 usec) 24.66 avg_batch 921 min_space
...

```

We're actually seeing output from both of our `pkt-gen` processes. One from the transmitting process, one from the receiving process. We manage to transmit 27 million packets per second (which adds up to nearly 13 gigabits per second). We only receive 3.7 million though. The remaining packets are lost.

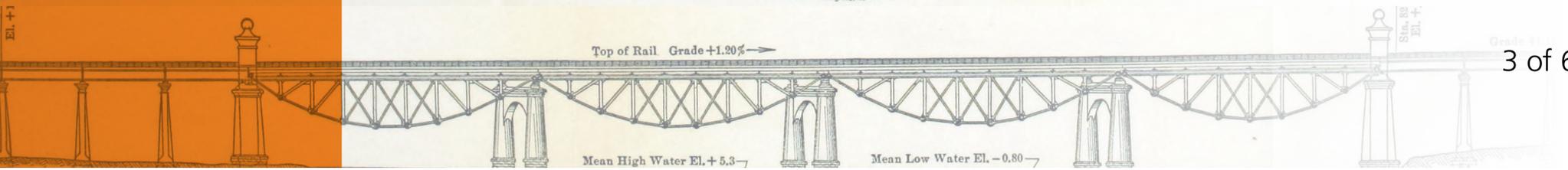
After 3.7 million brief moments of silence for those poor lost packets, we can ask why. If you remember only one thing from this article, remember that the most important question to ask when benchmarking is "Why?" "Why do we see that number?" "Why isn't it higher?" "Why isn't it lower?"

I happen to know, and you're just going to have to trust me, that if we'd route that traffic, we'd pass many more packets. That seems strange, because routing is harder work than switching.

Happily, FreeBSD has excellent tools that let us figure out where the system is spending its time. For this sort of work the `pmcstat(8)` tool is invaluable.

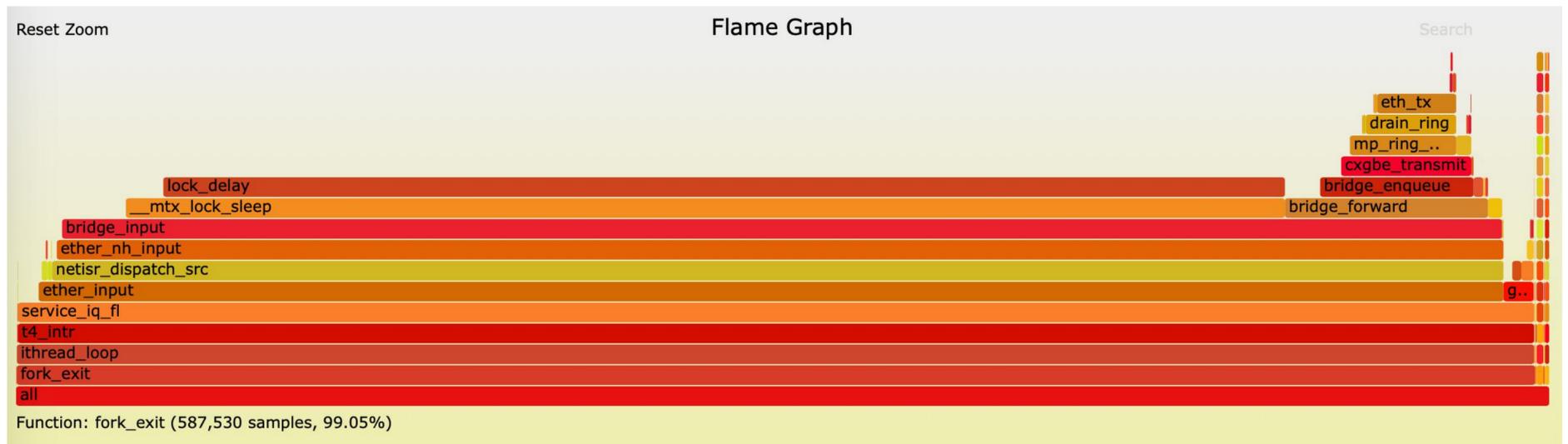
The following will produce a very instructive flame graph:

```
pmcstat -S cpu_clk_unhalted.ref_tsc -l 30 -z 50 -O data.out -q
pmcstat -R data.out -G data.stacks
stackcollapse-pmc.pl data.stacks > data.folded
flamegraph.pl data.folded > data.svg
```



The pmcstat tool is part of the base FreeBSD system. The stackcollapse-pmc and flamegraph tools are the work of Brendan Gregg (<https://github.com/brendangregg/FlameGraph>).

Anyway, pretty picture time:



This might look a little strange, but the most important thing it shows is where the system has been spending its time. The vertical axis shows a stack trace. Reading from the bottom it tells us that `fork_exit()` called `ithread_loop()` which called ...

Really, the bottom of the call stack isn't very interesting.

Let's work out what the system is doing when it could be passing around packets. Let's look for a function that seems to be taking a lot of time. The widest function--and thus the one where we spent most of our time—is `bridge_input()`. We spend 93% of our time there. That seems like it's related to bridging work. Why do we spend so much time in it? Most of the time we spend in `bridge_input()` we're actually doing `__mtx_lock_sleep()` rather than something useful-looking like `bridge_forward()`.

We shouldn't be napping. Why are we sleeping? And what's a 'mtx'?

Mutexes

`__mtx_lock_sleep()` is part of the mutex(9) code base. It's spelled 'mtx' in the function name because programmers are lazy. Mutexes provide mutual exclusion. They help ensure that only one CPU core can access a particular bit of memory at a time. Why is that required?

The simplest example is this: Imagine two CPU cores adding one to a number at the same time. Also imagine a universe where things always go wrong, where the toast always lands buttered side down. It may take some effort to imagine. No? You're there already? Fantastic.

So, we could have this order of operations:

```

CPU1: Load number (a = 1) from memory to a register (r0 = 1)
CPU2: Load number (a = 1) from memory to a register (r1 = 1)
CPU1: Increment register (r0 = 2)
CPU1: Store register (r0 = 2) to memory (a = 2)
CPU2: Increment register (r1 = 2)
CPU2: Store register (r1 = 2) to memory (a = 2)

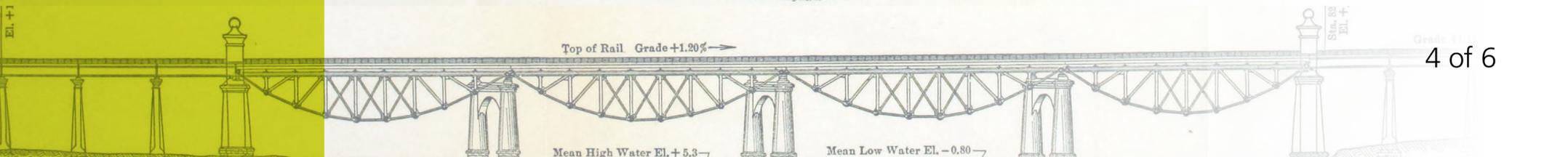
```

Whoops. Suddenly $1 + 1 + 1$ is 2. One way of resolving this problem is to ensure that CPU2 can't run this code at the same time, and that's what mutexes help us accomplish.

```

CPU1: Take mutex
CPU2: Try to take mutex. Fail and wait
CPU1: Load number (a = 1) from memory to a register (r0 = 1)

```



```

CPU1: Increment register (r0 = 2)
CPU2: Try to take mutex. Fail and wait
CPU1: Store register (r0 = 2) to memory (a = 2)
CPU1: Release mutex
CPU2: Take mutex
CPU2: Load number (a = 2) from memory to a register (r1 = 2)
CPU2: Increment register (r1 = 3)
CPU2: Store register (r1 = 3) to memory (a = 3)
CPU2: Release mutex

```

There are rather more complex flows in the `if_bridge(4)` code, but this gives an introduction to what mutexes are for.

The problem with mutexes is that while they ensure that the internal data structures can't be accessed by more than one CPU core at a time, that also means we can't do useful work on more than one core at a time.

It turns out that `if_bridge(4)` protects all of its internal data structures (such as 'this bridge connects interface `igb0` and `igb1`') with a single mutex, so no matter how many CPU cores you have only one of them can forward packets over the bridge at a time.

As a result, we can only forward about 3.9 million packets per second.

Improving This

One possible way of improving this is to change the mutex into a read/write lock (see `rwlock(9)`). Read/write locks make a distinction between read accesses and write accesses. If code is only going to read the data and not modify it, we can allow multiple CPU cores to do so at the same time. It's only when data will be modified that we need to deny access to all other CPU cores, even the ones that only want to read the data.

That would already improve matters greatly. A prototype along those lines improved throughput to about 8 million packets per second.

Doing Better

FreeBSD 13 (which is not released yet, so is now `CURRENT`) comes with a new way to handle this sort of problem. It's called `epoch(9)`, and was designed and implemented by people much, much smarter than I am.

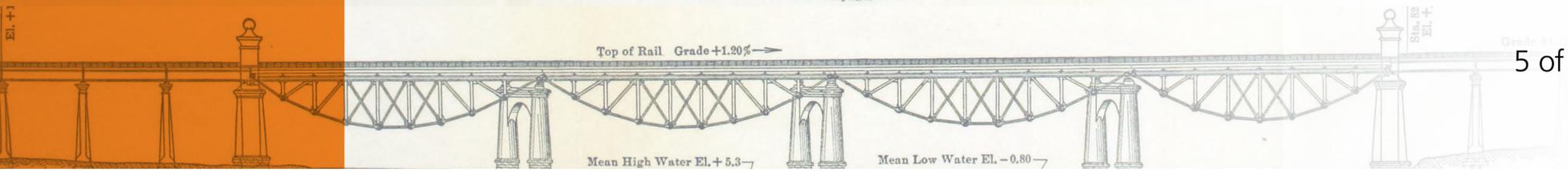
Let's pretend, for the purposes of this article, that I actually understand it, so I can explain what it does.

The `epoch(9)` facility originates from ConcurrencyKit (<http://concurrencykit.org>). When combined with the ConcurrencyKit lists (`ck_queue`), we can safely use protected data structures without acquiring a lock (either a mutex or a read/write lock) at all.

For `if_bridge(4)`, the main data structures are lists, and the typical changes are to add or remove an item to or from the list.

If we add something to the list, we need to ensure that the list itself is always in a valid state, but we don't actually care too much if a given CPU core sees the item in the list or not. This means that as long as we're careful to fully populate new data structures before we insert them in the list, we can safely insert them while other CPU cores are looking through the list.

When we need to remove something from the list, we do need to be more careful. We can remove the data from the list, but we can't actually delete it until we're sure no one is using it anymore.



That's what the epoch(9) facility is for. When we want to read the protected data, we inform the epoch(9) system through the `NET_EPOCH_ENTER()` wrapper, that we're going to be accessing this data. When we're done, we also let it know, through `NET_EPOCH_EXIT()`.

This allows the system to keep track of who could possibly still be accessing the data we're going to remove. Very simply put, what it does it keep track of when we tell it we want to delete something (by registering a callback to do the final cleanup, using `NET_EPOCH_CALL()` it waits until every CPU core that called `NET_EPOCH_ENTER()` has also called `NET_EPOCH_EXIT()`. At that point we can be sure no one is using the data anymore, and we can safely delete it.

There are some additional complications because the ConcurrencyKit lists are safe to read from while they're being modified, but we cannot have multiple CPU cores attempt to change them at the same time.

In `if_bridge(4)`, we use the existing mutex to protect write accesses to these lists. The final result is that we can still only perform one modification of the bridge state (e.g. adding an interface or learning which interface to use for a given MAC address) at a time, but we can keep processing packets on other CPU cores while we do this.

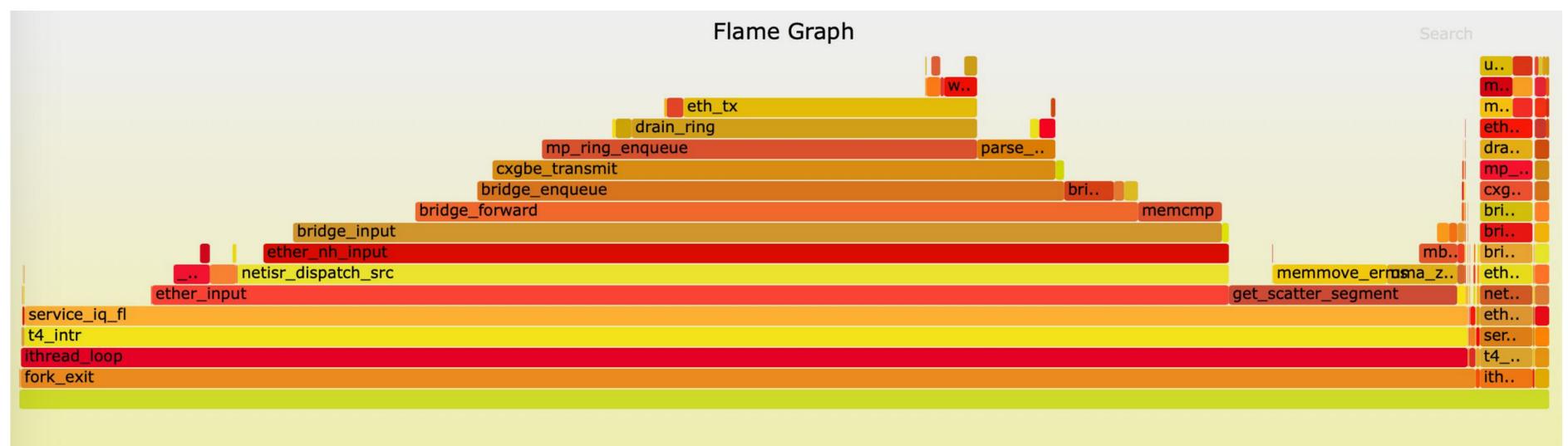
Remember that a single CPU core managed to process 3.7 million packets per second, so we can safely assume we can also handle many hundreds of thousands, or even millions, of changes to that state per second.

Final Measurements

With the epoch-based approach in place the performance test shows:

```
283.745634 main_thread [2666] 18.637 Mpps (18.673 Mppts 8.946 Gbps in 1001923 usec) 82.70 avg_batch 800 min_space
284.108710 main_thread [2666] 26.982 Mpps (27.009 Mppts 12.951 Gbps in 1001000 usec) 494.37 avg_batch 99999 min_space
284.746709 main_thread [2666] 18.620 Mpps (18.640 Mppts 8.938 Gbps in 1001076 usec) 82.80 avg_batch 816 min_space
285.172211 main_thread [2666] 26.981 Mpps (28.695 Mppts 12.951 Gbps in 1063501 usec) 494.47 avg_batch 99999 min_space
285.747709 main_thread [2666] 18.640 Mpps (18.659 Mppts 8.947 Gbps in 1001000 usec) 82.84 avg_batch 805 min_space
286.235212 main_thread [2666] 26.976 Mpps (28.676 Mppts 12.949 Gbps in 1063001 usec) 494.34 avg_batch 99999 min_space
286.748709 main_thread [2666] 18.638 Mpps (18.656 Mppts 8.946 Gbps in 1001000 usec) 83.01 avg_batch 800 min_space
287.236712 main_thread [2666] 26.976 Mpps (27.017 Mppts 12.949 Gbps in 1001500 usec) 494.41 avg_batch 99999 min_space
287.750709 main_thread [2666] 18.633 Mpps (18.671 Mppts 8.944 Gbps in 1002000 usec) 83.05 avg_batch 803 min_space
288.300211 main_thread [2666] 26.977 Mpps (28.690 Mppts 12.949 Gbps in 1063499 usec) 494.41 avg_batch 99999 min_space
...
```

So, we now forward about 18.6 million packets per second, or a 5x improvement. There's also an associated pretty picture (or flame graph):



Here we see that we're no longer spending all of our time (or indeed any of our time) in `__mtx_lock_sleep()`, but instead we're doing useful work. Almost all of our time is spent doing work directly related to working out where the packets need to go and sending them there.

Closing

This article discusses ongoing work, so if you find a bug: well done! Tell the author as he might not have spotted it yet.

For those of you interested, the pending patches can be found here:

- <https://reviews.freebsd.org/D24249>
- <https://reviews.freebsd.org/D24250>

This project was made possible by sponsorship from the FreeBSD Foundation.

KRISTOF PROVOST is a freelance embedded software engineer specializing in network and video applications. He's a FreeBSD committer, maintainer of the pf firewall in FreeBSD and a board member of the EuroBSDCon foundation. He has an unfortunate tendency to stumble into uClibc bugs and a burning hatred for FTP. Do not talk to him about IPv6 fragmentation.

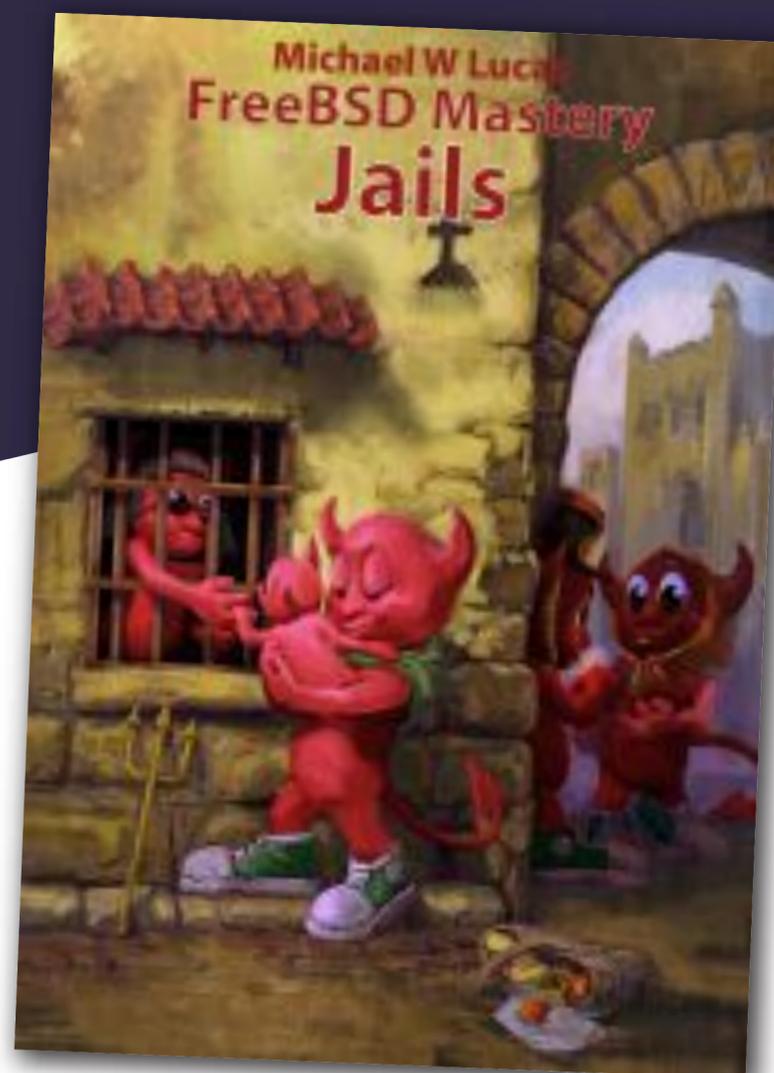
Jails ARE FBSD'S MOST LEGENDARY FEATURE:

KNOWN TO BE POWERFUL, TRICKY TO MASTER,
AND CLOAKED IN DECADES OF DUBIOUS LORE.

FreeBSD Mastery: Jails cuts through the clutter to expose the inner mechanisms of jails and unleash their power in your service.

Confine Your Software!

- * Understand how jails achieve lightweight virtualization
 - * Understand the base system's jail tools and the iocage toolkit
 - * Optimally configure hardware
 - * Manage jails from the host and from within the jail
 - * Optimize disk space usage to support thousands of jails
 - * Comfortably work within the limits of jails
 - * Implement fine-grained control of jail features
 - * Build virtual networks
 - * Deploy hierarchical jails
 - * Constrain jail resource usage.
- ...**And much, much more!**



FreeBSD Mastery: Jails BY MICHAEL W LUCAS **Available at All Bookstores**