SSH Bulk Transfer Performance

Allan Jude -- allanjude@freebsd.org

Introduction

- 15 Years as FreeBSD Server Admin
- FreeBSD src/doc committer (ZFS, bhyve, ucl, xo)
- FreeBSD Core Team (July 2016 - 2018)
- Co-Author of "*FreeBSD Mastery: ZFS*" and "*FreeBSD Mastery: Advanced ZFS*" with Michael W. Lucas
- Architect of the ScaleEngine CDN (HTTP and Video)
- Host of BSDNow.tv Podcast
- ZFS for large video collections, large website caches, mirrors of TrueOS pkgs and RaspBSD
- Single Handedly Manage Over 1000TB of ZFS Storage

Motivation

ScaleEngine uses SSH for bulk data transfer because it is the most convenient way to orchestrate the remote server receiving the data. We use ZFS replication for live failover, but also for pushing large sets of data to data centers all over the world.

ScaleEngine has four primary use cases for bulk data transfer:

ZFS Replication over LAN and MAN

We backup customer data between servers in our central data center, and the offsite locations. Data is usually “pulled” by the receiver. LAN connections recently grew to 10 gbps. MAN is a 1 gbps point to point link. Saturating a 1 gbps pipe even with a few milliseconds of latency is not hard.
We control both sides, and they are FreeBSD, so a modified client that is faster is possible.

ZFS Over Internet

This is used to publish specific datasets to remote servers, such as the TrueOS package repositories. Data is “pulled” by the receiver. Bandwidth-delay product works against us here. Getting higher speeds requires some work, and making this easier would be good.
Toronto to Germany is ~100ms.
Toronto to Melbourne is ~240ms.
Both ends FreeBSD, custom client ok.

Rsync

ZFS replication has replaced rsync almost everywhere. Except moving files from our Linux video transcoding rigs. Recordings of customers' live streams are recorded locally at various ingest servers around the world, then transferred to the central storage servers. This data is "pushed" from the recording servers to the storage servers.

Origin is Linux, prefer to avoid custom client.

Customer SFTP

Customers upload original copies of their video content to us via SFTP/SCP, and we want to offer the best possible upload speeds without requiring the customer to use a modified version of SSH. Who knows what OS they will be running, or what SSH client they will use. Mostly receiving, but care about both directions.

HPN

The HPN patches were first developed in 2004. The default SSH window size was 64 - 128 KB, which worked well for interactive sessions, but was severely limiting for bulk transfer in high bandwidth-delay product situations. The first patch enabled a dynamic window, allowing standard TCP window scaling, and offered much better transfer speeds over high latency links. The dynamic window feature only worked on HPN-to-HPN connections, so in other cases, the HPN patches increased the default window size to 2 MB.

With OpenSSH 4.7 in 2007, the stock default window size was also changed to 2 MB.

Manual Tuning

The HPN patches also added a client side configuration option, TcpRcvBuf, to manually specify a receive socket buffer size via setsockopt() SO_RCVBUF. This greatly increased transfer speeds when a client is receiving from a server. Performance for pushing data from a client to a server was still limited by the defined HPNBufferSize option, often suboptimal.

Bandwidth-Delay Product

Transfers over LAN are relatively fast but not quite able to saturation 10 gbps. But, what happens if you try to do it over the Internet?

Add even a mere 10 ms of delay, with a 4 MB socket buffer (double the default), and the theoretical maximum bandwidth drops to just 3300mbps. Netcat manages this, while stock SSH only gets ~160 mbps. HPN can receive 1300 mbps, but only send 175 mbps.

Reality is not 10 milliseconds

Bump the latency up to 100 ms, and the floor falls out from under you. With a 4 MB socket buffer, the theoretical capacity of the link is now just 335 mbps. Stock SSH manages between 9 and 14 mbps. HPN again can receive 180mbps, but only send the same 11 mbps as stock SSH. Even with a 32 MB socket buffer, stock SSH doesn't get any faster, because the SSH window is a fixed size. HPN can manage to receive 1000 mbps, but that is still much less than half of the theoretical ~2700 mbps BDP capacity.

Dynamically Not Scaling

ScaleEngine found it was necessary to manually set the HPN TcpRcvBuf settings to get acceptable transfer speeds. When this was investigated, it was determined that dynamic window scaling was not working. During both HPN and non-HPN bulk data transfers it was observed that the TCP window rarely grew beyond 256 KB.

Why Not?

When investigated, it was determined that the channel_check_window() function slides the SSH window forward each time half of the window has been consumed. In version 4.7 an additional check was added, and the window is slid forward if the consumed portion of the window exceeds 3 times the maximum packet size (32 KB in OpenSSH 7.2). We found that this pattern causes the TCP window to never increase much beyond that size, 128 KB.

What is Wrong?

The HPN patch dynamic window feature increases the maximum SSH window to 1.5 times the difference between the socket buffer and the maximum SSH window, but only if the socket buffer exceeds the maximum window size. Since this condition is never met, and the SSH window never grows, the TCP window never grows beyond half the size of the SSH window.

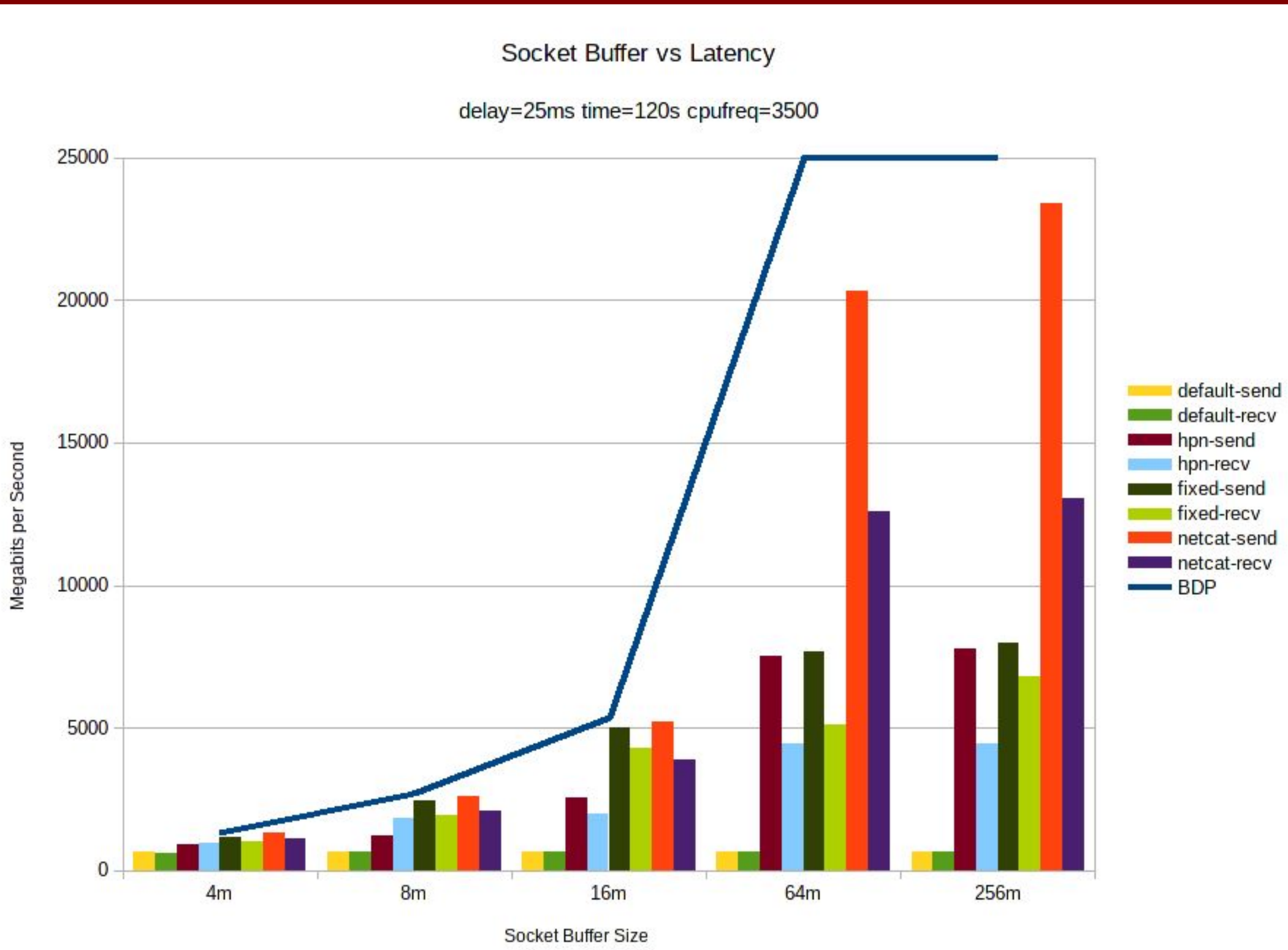
So, Fix it!

Our patch changes this behaviour to grow the SSH maximum window by 1.5 times the difference between the socket buffer and the unconsumed portion of the SSH window. This condition is now met once the TCP window grows to half of the maximum SSH window, and then the maximum SSH window is increased. The TCP window will grow further, to half of the new maximum. This process continues until the TCP buffer no longer needs to grow to maximize bandwidth, or the maximum size of the socket buffer imposed by the operating system is reached.

Results

With this fix in place, SSH can both send and receive at reasonable speeds even with a high bandwidth-delay product. The speeds achieved in testing were usually not much more than half of what netcat can do, but were still a very large improvement.

The change is restricted to non-interactive sessions, so the socket buffer of an interactive session will not grow and result in high input latencies.


Socket Buffer vs Latency
delay=25ms time=120s cpufreq=3500
Megabits per Second
25000
20000
15000
10000
5000
0
4m
8m
16m
64m
256m
Socket Buffer Size
default-send
default-recv
hpn-send
hpn-recv
fixed-send
fixed-recv
netcat-send
netcat-recv
BDP

Manual is Still Better

We find the TcpRcvBuf option from the HPN patches extremely useful. Rather than depending on the OS auto-scaling the socket buffer, we can just start with a large buffer immediately. This saves the first 2-4 minutes of the transfer being slow as the socket buffer grows.

Extending it Further

We needed this feature in the other direction, for sending to be faster. So we created a local protocol extension, RemoteRcvBuf. This allows the client to request that the server setsockopt() a larger receive buffer. The value is limited by a new SSH configuration option, and the limits imposed by the OS.

Tuning Tips

For SSH bulk transfer, it is desirable to avoid increasing the maximum size of the auto-scaling socket buffer, as this will impact all sockets on the system. The TcpRcvBuf feature, and its remote counterpart RemoteRcvBuf, allow the user to manually specify a larger static buffer for a single connection. You can tune the maximum buffer size to a very large value, allowing for extremely high bandwidth-delay products, while keeping the auto-scaling buffer at a reasonable size, to avoid consuming excess memory on a server that also serves many concurrent clients.

Sysctl City

This maximum size of an individual socket buffer is bounded by kern.ipc.maxsockbuf. This value is the maximum amount of memory that can be consumed by the buffer, not the maximum size of the buffer. 2048 bytes of buffer consumes 2048 bytes plus 256 bytes of overhead, so to support a 64 MB socket buffer, the maxsockbuf must be set to 72 MB.

- net.inet.tcp.{send,recv}space - initial size of the TCP socket buffer
- net.inet.tcp.{send,recv}buf_max - maximum size for auto-scaling
- net.inet.tcp.{send,recv}buf_inc - size of each growth increment
- net.inet.tcp.{send,recv}buf_auto - Enable/Disable auto-scaling
- kern.ipc.maxsockbuf - The maximum size of any socket buffer

Switching off the Crypto

The HPN patches also included a feature called the NONE cipher. This allowed a standard SSH session to be established, with encryption, then once the login process is finished, and the data transfer begins, the encryption was switched to a null cipher. The feature contains a number of protections to ensure it cannot be used for an interactive session, and can never spawn a shell.

HPN+NONE To the Rescue

Since 2011 ScaleEngine has made use of the HPN and NONE Cipher patches for SSH to accelerate ZFS replication, especially over LAN. Removing encryption and decryption from the pipeline made it possible to saturate 1 gbps interfaces with a modest CPU. The HPN patches improved performance of SSH over the Internet by using a larger sliding window.

Overcome by Events

The HPN patch doesn’t seem to help very much outside of manually requesting a large receive window. This only works if you are the receiver.

The NoneCipher is slower than some modern ciphers, because it still uses a MAC, the default UMAC64. This ends up being the bottleneck when trying to achieve 10 gbps.

So, what to do about it?

Lets look at where we are starting from

Mercat 5 and 6 @ Sentex

- Intel(R) Xeon(R) CPU E5-1650 v3 @ 3.50GHz
- 6 Cores + Hyperthreading (Turboboost Disabled)
- 32 GB RAM
- Chelsio T580-LP-CR 40 Gigabit NICs (back-to-back)
- FreeBSD 11.0-RELEASE-p1
- Base OpenSSH (default): OpenSSH_7.2p2, OpenSSL 1.0.2j-freebsd 26 Sep 2016
- HPN OpenSSH (hpn): OpenSSH_7.3p1, OpenSSL 1.0.2j-freebsd 26 Sep 2016
- Patched OpenSSH (fixed): OpenSSH_7.3p1, OpenSSL 1.0.2j-freebsd 26 Sep 2016

Measure First

Default cipher is ChaCha20-Poly1305

ChaCha20-Poly 1305: 1900 mbps

AES256/128-CBC: 2500/3000 mbps

AES256/128-CTR: 4900/5200 mbps

NoneCipher: 5800 mbps

AES256/128-GCM: 7800/8500 mbps

Netcat: 36000 mbps

Newer Crypto is Faster than NONE?

With modern hardware support for AES-NI, using the AES-GCM cipher is often faster than using the NONE cipher. Data is not encrypted, but a MAC is still applied, to detect modification of the data in transit. Whereas AES-GCM is an authenticated cipher and obviates the need to calculate a MAC as a separate pass. The fastest available MAC in OpenSSH is UMAC-64. On our test system, this limited the throughput of the NONE cipher to approximately 6,000 mbps, while AES128-GCM reached 9,000mbps.

None MAC!

We addressed this problem by developing a new feature, the NONE MAC. By switching to OpenSSL's null MAC, throughput up to 15,000 mbps was achieved. The same safeguards used for the NONE cipher are also applied to the NONE MAC. It cannot be used during an interactive session, or when a TTY is allocated. We do not require the protection of a MAC when doing ZFS replication, which does its own checksumming of the data.

Unclog the Pipe

Using the NONEMAC, that is no encryption, and no MAC, the patched version of OpenSSH was able to reach more than 80% of the performance of the netcat control transfer. AES-CTR was only ~10% slower than the NONE cipher, as both were constrained by the calculation of the MAC. The tests for AES-CBC and AES-CTR were then repeated with the NONEMAC. CBC mode saw 40% improvement for 128 bit, and 30% for 256 bit, while CTR mode results were improved by 90% and 80% respectively.

Fresh Numbers

AES256-CBC + NONEMAC: 3400

AES128-CBC + NONEMAC: 4500

AES256-GCM: 7800

AES128-GCM: 9000

AES256-CTR + NONEMAC: 9100

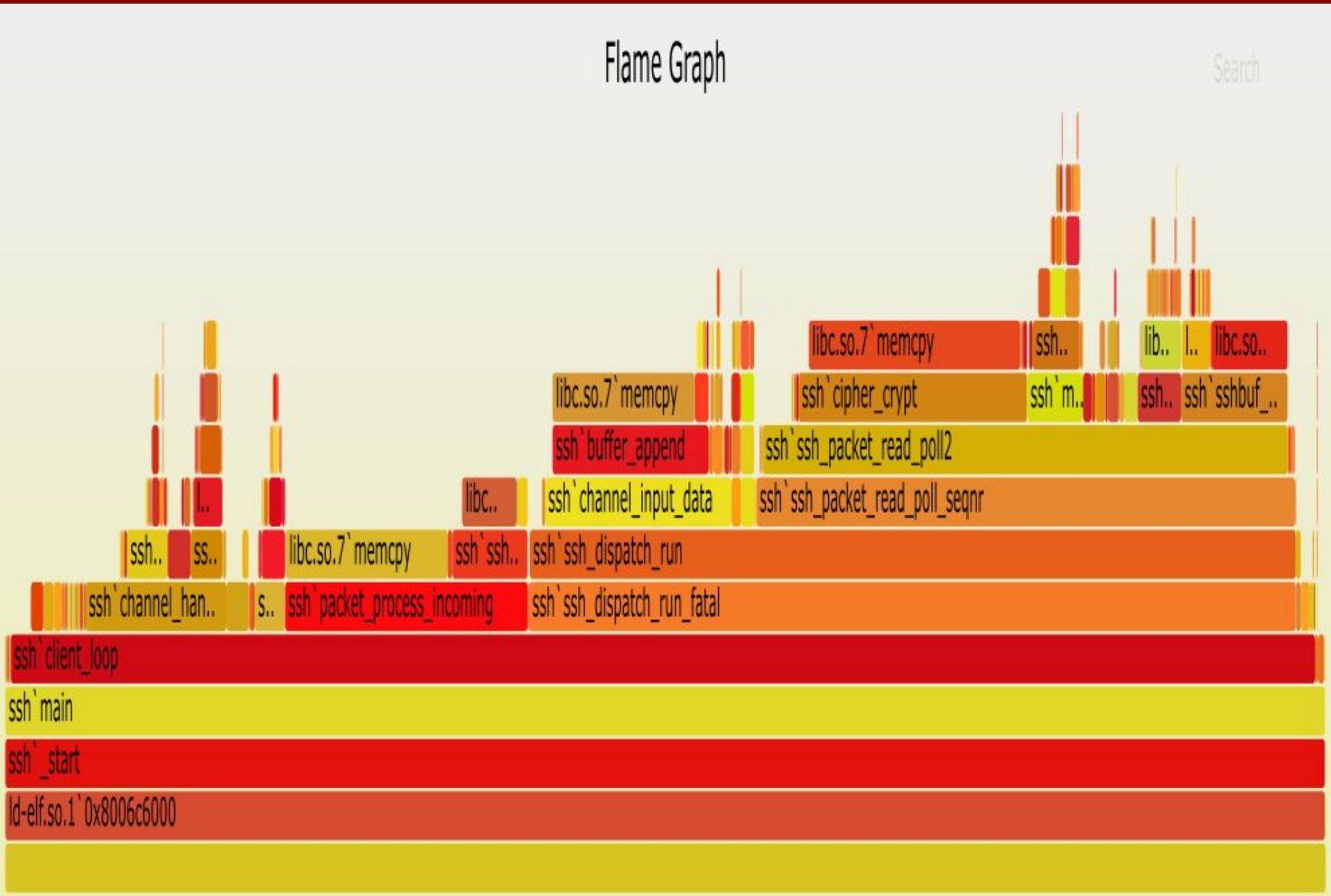
AES128-CTR + NONEMAC: 10100

NONE Cipher + NONEMAC: 13100

Netcat: 36000

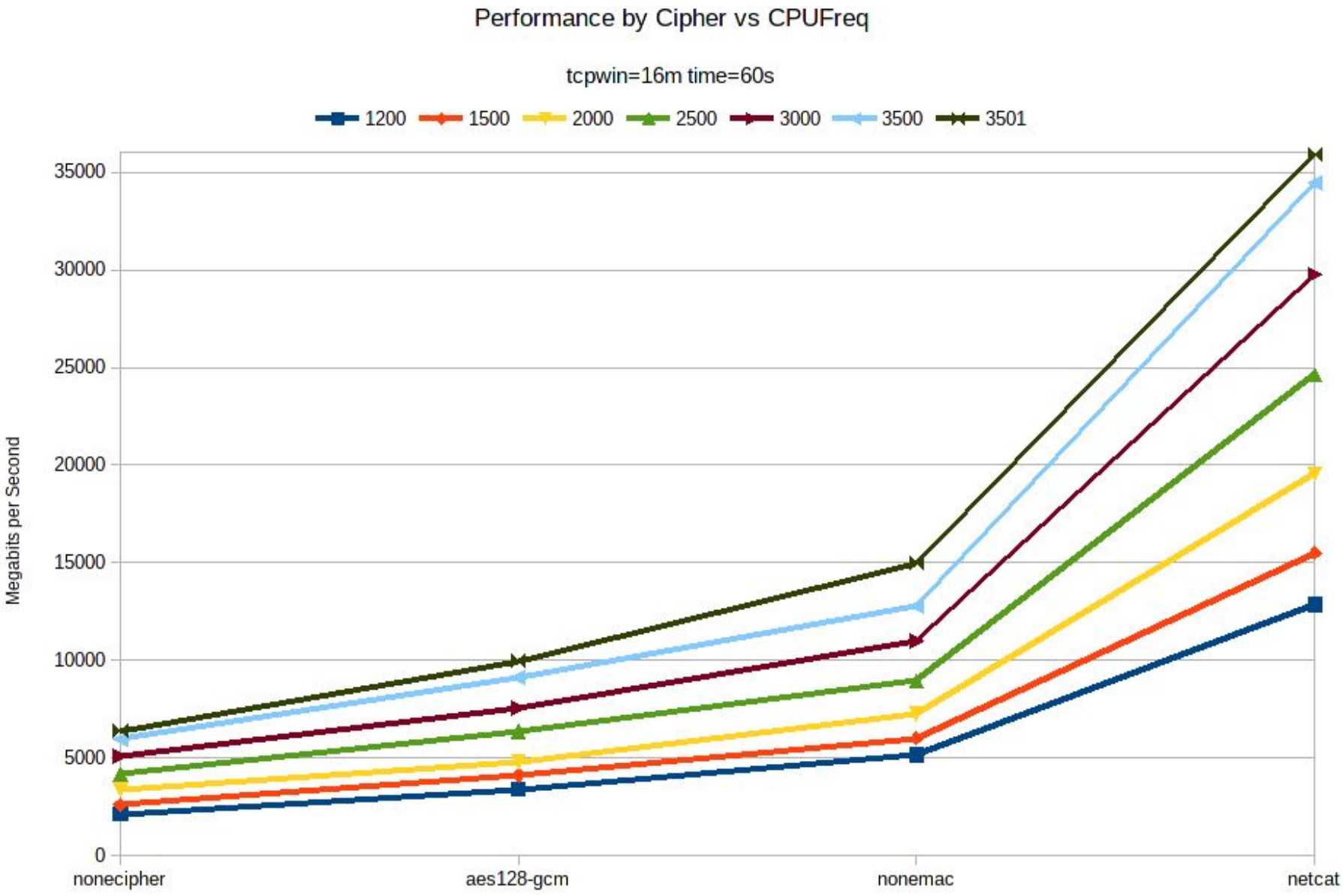
Limits of Tuning

At this point, this work has reached the limits of what can be achieved with minor patching and OS tuning. DTrace flame graphs (figure 7 and 8) show that almost all CPU time is now spent in libc (memcpy, memset, realloc, etc). In order to get more performance, it would likely be necessary to make architectural changes to OpenSSH, and this seems excessive considering the tool is already being abused much beyond its intended purpose.


Flame Graph
Search
libc.so.7`memcpy
ssh..
lib..
l..
libc.so..
libc.so.7`memcpy
ssh`cipher_crypt
ssh`m..
ssh..
ssh`sshbuf_..
ssh`buffer_append
ssh`ssh_packet_read_poll2
l..
libc..
ssh`channel_input_data
ssh`ssh_packet_read_poll_seqnr
ssh..
ss..
libc.so.7`memcpy
ssh`ssh..
ssh`ssh_dispatch_run
ssh`channel_han..
s..
ssh`packet_process_incoming
ssh`ssh_dispatch_run_fatal
ssh`client_loop
ssh`main
ssh`_start
ld-elf.so.1`0x8006c6000

CPU Scaling

Figure 9 shows that performances across all ciphers scales linearly with CPU clock frequency. Even netcat is constrained by the speed at which it can copy memory into the socket. Sadly this means that most Intel Xeon E5-26xx processors cannot yet saturate 10gbps network links, because of their lower relative clock speed compared to the E5-16xx processors used in the benchmarks.


Performance by Cipher vs CPUFreq
tcpwin=16m time=60s
1200
1500
2000
2500
3000
3500
3501
Megabits per Second
35000
30000
25000
20000
15000
10000
5000
0
nonecipher
aes128-gcm
nonemac
netcat