Measure Twice, Code Once: Network Performance Analysis for FreeBSD

George V. Neville-Neil
Neville-Neil Consulting
gnn@freebsd.org
Jim Thompson
Rubicon Communications
jim@netgate.com

February 27, 2015

Abstract

The networking sub-systems of any operating system have grown in complexity as the set of protocols and features supported has grown since the birth of the Internet. Firewalls, Virtual Private Networking, and IPv6 are just a few of the features present in the FreeBSD kernel that were not even envisioned when the original BSD releases were developed at U.C. Berkeley over 30 years ago. Advances in networking hardware, with 10Gbps NIC cards being available for only a few hundred dollars, have far outstripped the speeds for which the kernel's network software was originally written. As with the increasing speed of processors over the last 30 years, systems developers and integrators have always depended on the next generation of hardware to solve the current generation's performance bottlenecks, often without resorting to any coherent form of measurement. Our paper shows developers and systems integrators at all proficiency levels how to benchmark networking systems, with specific examples drawn from our experiences with the FreeBSD kernel. Common pitfalls are called out and addressed and a set of representative tests are given. A secondary outcome of this work is a simple system for network test coordination, Conductor, which is also described. The Conductor system, as well as all the tests and results are published, in parallel, in two open source projects and (http://github.com/gvnn3/conductor) and (http://github.com/gvnn3/netperf)

1 Network Benchmarks are Difficult

Reading over any company's published benchmarks fills any competent engineer with skepticism, as it should. Creating a broadly applicable and honest benchmark is difficult work, and many benchmarks are not meant to be unbiased. Some are quite the opposite, meant to show the author's system in the best possible light. Often a benchmark is picked and promoted because it is the first one that showed what the author thought of as good performance for the system being measured.

The technical challenges in developing a benchmark aren't limited to avoiding lying to the reader. Figuring out how to measure something reliably and repeatably in a dynamic system requires a deep understanding of all of the system components and how interactions among them can conspire to create false measurements. A non-networked system, where the interconnections and interactions among the various components are, or should be, visible, is still complex enough to trip up many developers. A networked system is far harder to measure for several reasons, including: asynchrony, visibility and the lack of well synchronized clocks.

Unlike other aspects of computer systems, networks are understood to be unreliable, with higher layer protocols providing features, such as reliability, and ordering, that software running on a single host takes for granted. The acceptance of unreliability at many layers of a network stack make it difficult to measure performance from only one viewpoint, either the sender or receiver. To understand networking performance we must observe at several points in any system, the sender, the receiver and, depending on the network application, any systems that exist between the two endpoints. When an operation is measured on a single host we can use the host's clock to measure the time that the operation took to complete. The difficulty of synchronizing clocks across a network means that we do not have as reliable a way to measure operational latency as if the software were on a single host. Recording the time that a packet was sent on one host and then again on another host when the packet is received is only as reliable as the level of synchronization between two hosts. Two, or more, hosts can be synchronized to within a millisecond using the Network Time Protocol (NTP) or to within a microsecond using the Precision Time Protocol (PTP) but most networked systems cannot use PTP in large deployments, and, even if they could, a 10Gbps network, which is common in a modern data-center, has latencies that are measured in the low tens of microseconds, making accuracy between hosts hard to determine.

Accurate timing is just one problem facing the engineer who has to perform a network benchmark. The first problem is simply understanding what is the right thing to measure. Some applications are latency sensitive, some sensitive to loss, and others require large amounts of bandwidth. Working out which of these three axes matters most to the application is the primary task that must be done before a benchmark can be begun. A full treatment of this and other performance measurement topics is the subject of an excellent book [3].

In this paper we will discuss some very basic bench-marking techniques and show how they were carried out on several systems, first looking at packet forwarding in FreeBSD[2], and then expanding to show a comparison of four firewalls. The goal of this work is not to be the definitive result on a particular measurement, but to show how we constructed our benchmarks and how the same techniques can be applied to measure and improve network performance.

2 Understanding Modern Hardware

Networking hardware long ago passed speeds of 1 Gbps with 10Gbps and higher being within easy reach of even the smallest companies. While network speeds have continued to climb the speeds of processors have topped out at around 3GHz and memory buses have stayed closer to processor speed increases. These speed mismatches have led to the creation of systems with architectures that attempt to split up work among many similar parts to get a single job done. Multiple cores, memory buss-es, and network queues all contribute to measurement noise when bench-marking on modern hardware.

To achieve top performance all of the system's resources, including CPU cores and caches, memory, and network queues must be in alignment. The resources are in alignment when packets with the same four-tuple of `Source IP, Destination IP, Source Port, Destination Port` are always handled by the same core, cache and queue. Why is this this alignment necessary?

High speed networks, which is currently any network that has a 10Gbps or higher bandwidth, handle packets at such a high rate that there is very little time for the CPU to do any work with the packet. At 10Gbps, minimum sized packets, which contain 64 bytes, arrive at approximately 14 million per second. The system receiving these packets has 67 nanoseconds to process each one. In this time frame, a 3GHz processor has approximately 200 cycles in which to process each packet. This seems adequate until the cost of accessing memory and caches is taken into consideration. A single cache miss can cost 32ns which means that every time a packet arrives and the relevant data to handle that packet is not on the core that the packet ultimately is routed to the system needs to evict data from the cache and then retrieve data from memory. Data relevant to the handling of a packet includes forwarding and routing entries, socket buffers, and all of the kernel's locking data structures related to the packet. The cost of cache misses is now high enough that when working with high speed packet flows any such process *must* be set to a single CPU core in order to achieve its highest performance. Allowing the scheduler to decide where the process should best be executed results in swings in the packet forwarding rate as well as an overall lowering of performance.

The only way to obtain the full performance out of high speed NICs and modern hardware is to be aware of the complete system topology. Packets must arrive and leave on a network card that is on the a bus connected to the CPU that has the application running on it. If an application is moved among cores and therefore loses cache locality, performance will suffer. On FreeBSD this means that network applications *must* be tied to a processor core using the `cpuset` program, or system calls, in order to work at top speed. Not tying a networking program to a core is the most common mistake that engineers make when working with high speed network hardware.

Multi-socket systems, those with two or more multi-core CPUs, exacerbate this problem because of the high cost of inter-processor communication. Any Intel based multi-socket system built since the advent of QPI, which includes all systems built in the last two years, connects each device, such as a network controller, to at most one CPU in the system. When a 10Gbps NIC is plugged into the bus it will be close to only one of the CPUs. If an application handles packets on the NIC from the foreign CPU the penalty is so high that there is no reason to bother with the extra cost of additional CPUs.

3 Test Automation

To undertake our work we created a simple coordination program and protocol, Conductor, that can be used to setup, run and reset systems without manual

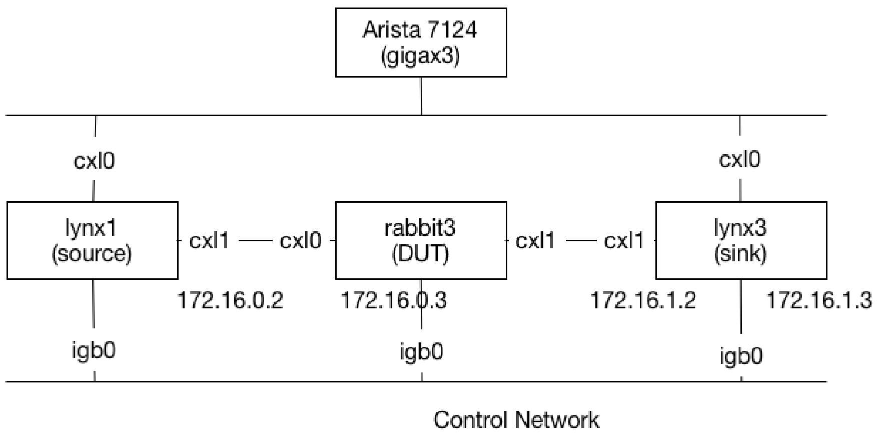

Figure 1: Lab Setup

intervention. The Conductor system is a small set of Python libraries that are used to coordinate various systems during testing. A typical test setup can be seen in Figure 1 where three hosts are used to test the forwarding of packets. The host *lynx1* sends packets to *rabbit3*, the *DUT*, via rabbit3's *cxl0* interface. The packets are forwarded by rabbit3 out of its *cxl1* interface where they are counted by *lynx3*. Conductor is used to coordinate these processes. Running from a master host, *zoo*, which resides on the administration network and is not shown in the diagram, Conductor contacts a `player` program on each host and executes commands, in four phases, to run test scenarios in a reproducible way.

The *Setup* phase is where device drivers are loaded, route entries are set up, and initial `ping` commands are used to make sure that the `ARP` tables are prepared for the test. One the setup is complete the *Run* phase starts up the packet generator sources and sinks. Commands can be given a timeout so that programs that do not have a built in timeout can be forced to exit after a pre-programmed number of seconds. The *Collection* phase copies off the results of the tests, including log files, packet captures, and any other output related to the test. Finally, the *Shutdown* phase removes the routes and the drivers loaded into the kernel during the *Setup* phase to bring the systems back to their initial, pre-test, state.

4 Kernel Packet Forwarding

One area of the FreeBSD operating system that is ripe for both performance analysis and improvement is the forwarding of packets via the base network stack as well as through various firewalls. There are several reasons that packet forwarding is a good candidate for performance work. First, there is a simple, quantitative, measure of performance, packets per second. Secondly the workload is both generated and measured outside of the system that is being tested, which we refer to as the Device Under Test (DUT). The external generation and reception of the packets means that the DUT is *only* doing the work that we care about in our testing, rather than using up cycles with overhead.

```
0.00-1.00  sec 1.09 GBytes 9.41 Gbits/sec
1.00-2.00  sec 1.10 GBytes 9.41 Gbits/sec
2.00-3.00  sec 1.10 GBytes 9.41 Gbits/sec
3.00-4.00  sec 1.10 GBytes 9.41 Gbits/sec
4.00-5.00  sec 1.10 GBytes 9.41 Gbits/sec
5.00-6.00  sec 1.10 GBytes 9.42 Gbits/sec
6.00-7.00  sec 1.10 GBytes 9.41 Gbits/sec
7.00-8.00  sec 1.10 GBytes 9.41 Gbits/sec
8.00-9.00  sec 1.10 GBytes 9.41 Gbits/sec
9.00-10.00 sec 1.10 GBytes 9.41 Gbits/sec
```

Figure 2: Baseline TCP Performance with iperf3

Since the first versions of BSD the operating system has had the ability to act as a router as well as an end system, forwarding packets, often as a small office router. FreeBSD also includes two, well known, firewall subsystems. *IPFW* was originally written for FreeBSD by Luigi Rizzo [4] and has been enhanced and extended over the last twenty years. The *PF* firewall was developed by OpenBSD and then ported to FreeBSD. *PF* was not written for an *SMP* kernel but was updated, by Gleb Smirnoff and others, to run in the FreeBSD kernel without a giant lock. *IPFW* and *PF* do not have the exact same features but they both implement a firewall which can either pass or drop traffic between interfaces.

Before any measurements can be made of the firewall software, a baseline needs to be taken of the systems being used. We undertook several experiments to establish how quickly the kernel could forward packets without subjecting them to the work required by a firewall, or the kernel itself. Our experiments were carried on on three systems in the Sentex Test Lab, lynx1, lynx3, and rabbit3. Lynx1 acted as the traffic source, sending packets via rabbit3 on to lynx3. Both of the lynx machines are dual socket with 10 core E5-2680 Xeon processors clocked at 2.8Ghz. Rabbit3 is a single socket, four core, E5-2637 Xeon, running at 3.GHz. Lynx1's second 10G port was connected directly to Rabbit3's first 10G port and Lynx3's second 10G port was connected directly to Rabbit3's second 10G port as can be seen in Figure 1.

One of the easiest and most common network tests is a single stream of TCP packets. Using the `iperf3` program we established a simple baseline for TCP within our test lab. Each test was carried out over 10 seconds, and showed little or no variance in bandwidth. The output shown in Figure 2 is `iperf's` output, reporting bandwidth every second for the entire 10 second run.

The output shown in Figure 2 is `iperf's` output, reporting bandwidth every second for the entire 10 second run.

There are many reasons why one should not only use TCP to test networking equipment. Unlike other protocols, TCP attempts to use the underlying network as efficiently as possible and it will adjust its behavior in the face of errors. While this is good for normal users, it will not show the rougher edges of a piece of networking software or equipment. As a simple baseline, it is a good place to start, but it is misleading as this one test might lead us to believe that the system is capable of forwarding 10Gbps in all situations, which is not the case.

```
827.257743 main_thread [1512] 14697768 pps (14726664 pkts in 1001966 usec)
828.259812 main_thread [1512] 14668997 pps (14699332 pkts in 1002068 usec)
829.261742 main_thread [1512] 14695277 pps (14723654 pkts in 1001931 usec)
830.263743 main_thread [1512] 14685547 pps (14714933 pkts in 1002001 usec)
```

Figure 3: pkt-gen output, source side, 64 byte packets

```
866.466039 main_thread [1512] 11943109 pps (11955136 pkts in 1001007 usec)
867.468024 main_thread [1512] 11946111 pps (11969824 pkts in 1001985 usec)
868.469126 main_thread [1512] 11942020 pps (11955180 pkts in 1001102 usec)
869.471027 main_thread [1512] 11939957 pps (11962655 pkts in 1001901 usec)
```

Figure 4: pkt-gen output, destination side, 64 byte packets

When testing systems that are responsible for forwarding packets, such as switches, routers and firewalls, we want to measure the number of packets which can be carried across the DUT. Such tests are carried out using network layer packets where the transport protocol, such as TCP or UDP, is mostly irrelevant.

Before passing packets through the DUT we established a raw baseline between two cards, connected back to back. A 10G network can theoretically move just over 14 million 64 byte packets per second. We wanted to establish just how many packets our systems could send and receive before adding in any other software layers. Using the pkt-gen program from netmap[5] we sent 64 byte packets from lynx1 to rabbit3 over a back to back link. A portion of the output at the source is shown in Figure3 and at the destination in 4. We used pkt-gen because it, in turn, uses the `netmap(4)` driver, allowing us to send and receive packets at near line rate.

The initial test shows that minimum sized Ethernet frames, those of 64 bytes, cannot be completely absorbed by the DUT on its 10G interface. Running a series of packets between the source and DUT shows that packets of 128 bytes or longer are all properly received by pkt-gen. The results of a full sweep of packet sizes are shown in 1. By using pkt-gen and netmap we are able to write packets directly to, and read them directly from the card without the packets going through any of the operating system's network processing software, thus giving us a good baseline measurement of what the underlying hardware and drivers can achieve.

Once we had established baseline for our cards and hosts we then measured the ability of the operating system to forward packets between interfaces. In

¡¡¡¡¡¡¡ HEAD ======= ¿¿¿¿¿¿¿ refs/remotes/origin/master

Size	TX	RX	Stddev
64	14,644,664	11,943,632	2366
128	8,215,485	No Loss	N/A
256	4,464,323	No Loss	N/A
1024	2,332,123	No Loss	N/A
1500	820,229	No Loss	N/A

Table 1: Card to Card UDP Baseline

Size	TX	RX	Stddev	% Line Rate
64	14,685,502	1,069,691	165	7
128	8,215,485	1,051,849	177	14
256	4,464,323	952,227	154	21
512	2,332,123	949,432	165	41
1024	1,192,770	948,172	100	80
1500	820,229	820,215	1.44	100

Table 2: IP Packet Forwarding

the forwarding scenario a single stream of UDP/IP packets was sent from lynx1 through rabbit3 and on to lynx3 where they were counted. No software other than the operating system was running on rabbit3, and its only job was to forward the packets between the cxl0 and cxl1 interfaces seen in Figure 1.

Our second set of measurements were carried out with a variety of packet sizes, including: 64, 256, 512, and 1500 byte packets. Each measurement was run over 30 seconds and then the data was run through ministat to find the median forwarding rate. Jumbo frames, which are packets of greater than 1500 bytes, were not tested. Packet forwarding is measured in packets per second (PPS) which is related to raw bandwidth by multiplying the PPS by the packet size.

The results of our packet forwarding experiment are given in Table 2. We observe that once the operating system becomes involved in the forwarding of packets our PPS measurements drop precipitously for all but full sized Ethernet frames. With minimum sized frames, those of 64 bytes, the operating system kernel is only able to forward seven (7) percent of the packets that are sent to it. There are few networks that deal only in minimum sized frames, as as such frames can hold, at most, 10 bytes of data when UDP is used with IPv4. Minimum sized frames are normally seen only as TCP ACK packets, which contain no data. Networking tests often use minimum sized frames to show how well a piece of software or hardware handles the worst possible scenario. As the packet sizes increase the system is eventually able to forward packets at line rate.

A packet forwarding test such as this one shows us an important aspect of networking benchmarks, that raw bandwidth is not always the best measurement of network performance. It is important to know how much overhead there is in the measurement. The smaller the packet, the more overhead there is for the amount of data transferred. A system that is forwarding minimum sized packets, at line rate, is still forwarding less actual data, than a system that is forwarding full sized packets at line rate. A 10Gbps network with 64 byte packets is transporting about 1.1Gbps of data because each 64 byte packet has 58 bytes of packet headers, 14 for Ethernet, 20 for IPv4, and 20 for TCP and a 4 byte checksum. With 58 bytes of overhead per 64 byte packet, 6.8Gbps of the bandwidth are being used for packet headers along, with 10 bytes left over for actual data when using UDP, with TCP there are only 6 bytes remaining for user data.. When 1500 byte packets are used, 9.4Gbps of the bandwidth is usable data and only 354Mbps is given over to overhead.

Size	TX	RX	Stddev	% Line Rate
64	14,685,502	1,093,090	634	7
128	8,215,485	1,079,852	549	14
256	4,464,323	1,273,975	141	28
512	2,332,123	1,267,776	136	54
1024	1,192,770	1,192,755	11595	100
1500	820,229	820,215	2.08	100

Table 3: IP Fast Packet Forwarding

```
ether_input()
netisr_dispatch_src()
ether_nh_input()
ether_demux()
netisr_dispatch_src()
ip_input()
ip_forward()
ip_output()
ether_output()
```

Figure 5: Normal IP Packet Forwarding Callgraph

4.1 Why?

One reason that forwarding packets via the operating system kernel is so slow is that each packet that flows through `ip_input()` and `ip_forward()ip_output()`, the kernel's packet input, forwarding and output routines, is subjected to a very large number of checks before the packet is handed back to the outgoing network device.

The call-graph for the normal packet forwarding path through the kernel, between `ether_input()` and `ether_output()` is shown in Figure 5. The output in Figure 5 was gathered by using DTrace[1] to show the stack above `ether_output()` at the point where that routine is called during packet forwarding. When a packet is forwarded via the kernel, each packet is subjected to several tests within each of the functions shown, and each of these functions adds overhead to ever packet that is forwarded.

To improve the packet forwarding performance of the system, a fast path routine was added in 2003, `ip_fastforward()`, in which many of the checks are deferred until after we know we can forward a packet. Any packet that was nominally correct and was not bound for the local machine was forwarded without being subjected to further checks. In a forwarding situation many of the attributes that `ip_input()` and `ip_forward()` would check for, such as packet options, never occur. Deferring these expensive operations after a point at which a typical packet would be forwarded increases overall performance by lowering the per packet overhead.

Using `DTrace` we are able to measure the time a packet takes to traverse the system by recording the time when it arrives in `ether_input()` and leaves via `ether_output()`. A D script was run for 30 seconds to record the time between these two calls. During those 30 seconds the DUT was subjected to a stream of 512 byte, UDP/IP, packets, which is the size of packet that showed

```
ether_input()
netisr_dispatch_src()
ether_nh_input()
ether_demux()
ip_fastforward()
ether_output()
```

Figure 6: Fast Packet Forwarding Callgraph

```
value  ------------- Distribution ------------- count
512 |                                          0
1024 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@  1414505
2048 |@                                         35478
4096 |                                          481
8192 |                                          0
```

Figure 7: Nanoseconds per packet, normal forwarding

a large difference between the normal and fast forwarding cases in Tables 2 and 3. Figure 7 shows a histogram of the times between `ether_input()` and `ether_output()` without fast forwarding, the baseline case. Figure 8 shows the same script with the same packet stream with fast forwarding turned on. Note that far more of the times fall into the 1024 bucket than into any of the others. The lowered per packet processing time leads directly to the higher PPS shown in Table 3.

5 A Test of Firewalls

Several sets of measurements were carried out to compare various fire-walling solutions. Four software stacks were tested: FreeBSD 11-CURRENT, pfSense 2.2 (based on FreeBSD 10.1), OpenBSD 5.6, and CentOS 7. FreeBSD, pfSense and OpenBSD all used the pf packet filter and CentOS 7 used Linux's iptables. All of the measurements were carried out on different hardware than that used in Section 4. For this set of tests hardware that is normally used for corporate firewalls was chosen, to give a more realistic example of what can be achieved with open source firewall software. The DUTs are C2758, 1U systems, equipped with a single, 8 core, 2.4GHz Atom processor, 8G of RAM and an Intel 82599 based 10G network interface. The source and sink systems were Xeon HC boxes X5680 @ 3.33GHz with Intel 82599 based 10G network interfaces running with Linux and DPDKs pktgen on the source, and FreeBSD 11-CURRENT (NODEBUG kernel) on the the sink. A slower box was intentionally chosen for the DUT to ensure that the source could outperform it and place it under sufficient stress during testing.

The DUT was placed between the source and sink in back to back fashion, similar to what is seen in Figure 1 with lynx1, rabbit3 and lynx3. Operating system kernels were chosen such that they did not have expensive debugging overhead included in them. In particular, on FreeBSD 11, which is the *head* of the tree under active development, the `GENERIC-NODEBUG` kernel was used

```
 value  ------------- Distribution ------------ count
512 |                                         0
1024 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@  1721837
2048 |@                                        41287
4096 |                                         490
8192 |                                         0
```

Figure 8: Nanoseconds per Packet, Fast Path

```
set limit states 100000000
match out on ix0 from 198.18.0.0/24 to any nat-to ix0
pass in quick all keep state
pass out quick all keep state
```

Figure 9: OpenBSD pf Rule-set

so that features that help with debugging SMP systems, such as the `WITNESS` system for tracking lock order reversals are turned off. A series of packets was then sent through each DUT and measured at the sink host.

The rule sets used for each test were created to be nearly identical in function. The OpenBSD rule set is shown in Figure 9, while the pfSense and FreeBSD rule sets, which are identical, are shown in Figure 10. The Linux rule set is shown in Figure 11. Each rule set sets up the DUT to perform network address translation on packets that are transiting the system via the 10G interface. Because NAT requires the system to modify every packet that passes through the DUT it is a good test of the performance of the software stack.

The first set of measurements tested the DUT's ability to forward packets using only a single core of the eight available in the host with the NAT filtering disabled. Forwarding packets without NAT gives us a baseline to measure against in the following test. The results shown in Table 4 give an idea of the per core, packet forwarding power of the DUT. In this set of tests pfSense fared best and Linux worst, with pfSense forwarding more than twice as many packets as Linux, and twice as many as a standard installation of FreeBSD 11. pfSense is built as a firewall, rather than as a general purpose operating system, and we can see here that it is well tuned to its job.

With the filtering turned on the packet rates drop, as seen in Table 5, as each packet filter is inspecting and modifying each packet that it sees. We see that pfSense remains ahead of all the other systems and that FreeBSD 11 drops to last, just behind Linux.

A single core may be an interesting way to establish a scaling target, that

```
set limit states 100000000
nat on ix0 from 198.18.0.0/24 to any -> ix0
pass in quick all keep state
pass out quick all keep state
```

Figure 10: FreeBSD and pfSense Rule-set

```
iptables -t nat -A POSTROUTING -j MASQUERADE -s 198.18.0.0/24 -o enp4s0f0
iptables -t filter -A FORWARD -j ACCEPT -m conntrack --ctstate ESTABLISHED,RELATED
iptables -t filter -A FORWARD -j ACCEPT
```

Figure 11: Linux iptables

¡¡¡¡¡¡¡ HEAD ======= ¿¿¿¿¿¿¿ refs/remotes/origin/master

OS Version	PPS	StdDev
pfSense 2.2	494,224	1944
OpenBSD,5.6	360,147	1162
FreeBSD 11-CURRENT	249,464	498
CentOS 7	198,239	172

Table 4: Single Core Performance w/o Filtering

OS Version	PPS	StdDev
pfSense 2.2	228,558	1440
OpenBSD 5.6	187,523	78
CentOS 7	139,797	95
FreeBSD 11-CURRENT	131,795	229

Table 5: Single Core Performance with Filtering

OS Version	Multi-Core	Single Core	Speedup
CentOS 7	945,807	198,239	4.7x
pfSense 2.2	920,415	494,224	1.8x
FreeBSD 11-CURRENT	684,721	249,464	2.4x
OpenBSD 5.6	361,253	360,147	N/A

Table 6: Multi-Core Performance w/o Filtering

OS Version	Multi-Core	Single Core	Speedup
CentOS 7	681,980	139,797	4.8x
pfSense 2.2	505,417	228,558	2.2x
FreeBSD 11-CURRENT	282,163	131,795	2.1x
OpenBSD 5.6	186,865	187,523	N/A

Table 7: Multi-Core Performance with Filtering

is, in a perfect world, if a single core can forward N packets then M cores can forward $M * N$ packets. In real world situations this is not the case due to numerous factors, including scheduling overhead, cache pollution, interrupt routing and others which will be covered in future work. In Table 6 we see what each system can forward when using all the available CPU cores with filtering turned *off*. With multiple cores we see that Linux and pfSense shoot far ahead of OpenBSD and that FreeBSD also lags behind the two leaders.

Looking at Table 6 we are struck by the fact that OpenBSD does not show any increase in performance. The reason for this poor showing is that OpenBSD remains a single threaded kernel, with a single lock protecting all kernel data structures, and therefore cannot take advantage of the extra cores in the system.

The final set of firewall tests shows the performance of each system while filtering packets using all available cores. As before OpenBSD does not gain any advantage from multiple cores and even falls slightly behind its single core performance, possibly due to caching effects with multiple cores, and caches come into play. CentOS increases its performance over filtering on a single core by roughly the same amount as it increased when not filtering packets 4.7x and 4.8x speedup respectively. The pfSense system increase its packet rate considerably more when filtering than when not filtering (1.8x and 2.2x).

A full comparison of all the results can be seen in Figure 12. The top of each bar gives the measured packets per second while the labels along the X axis indicate whether the measurement was taken with filtering turned on or off, as well as whether the DUT was using one or more cores.

6 Conclusions

As was stated in Section 1, benchmarks are difficult to design and to evaluate. Modern hardware, with multiple cores, differing cache, memory and device hierarchies make the process of getting reliable numbers out of a set of benchmarks that much more difficult. Furthermore, tools to extract performance information from computer systems, such as on chip hardware performance monitoring counters, Intel's pcm and VTune tools, and others, provide either too much or too little data, and all suffer from a *probe effect* whereby their use unduly influences the DUT.

In this paper we chose a very simple workload, packet forwarding, for several reasons. The first is that the measurement tools are completely outside of the DUT, and reside on boxes where we do not care about the performance of the system so long as it is capable of generating and absorbing the required number of network packets. Keeping the tools outside the DUT thereby reduces the number of variables we have to control for on the system. Secondly, the measurement, packets per second, is easy to understand from a quantitative

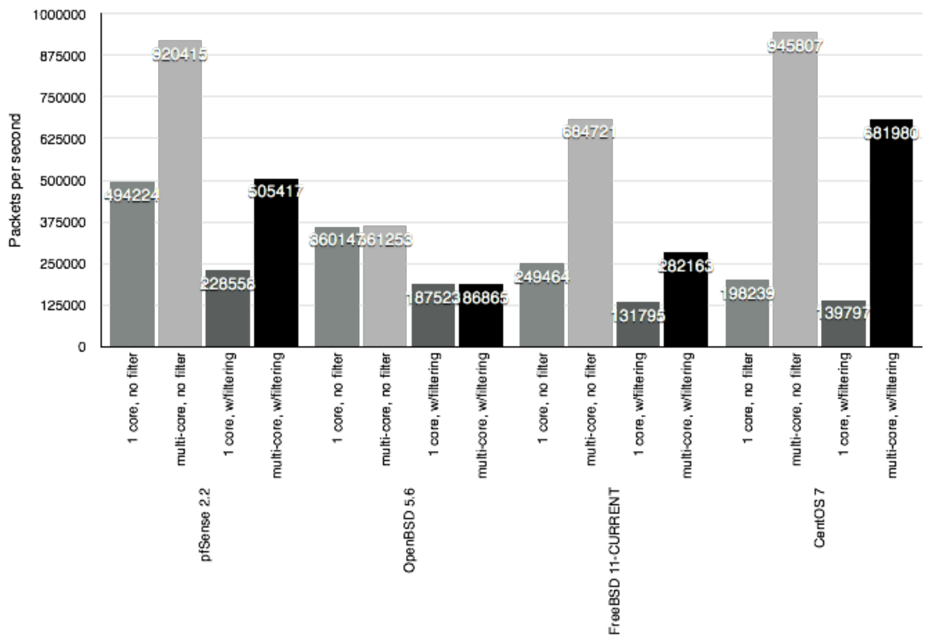

Figure 12: Complete PF Comparison

perspective, more is better, which makes comparing various systems, such as the firewalls in Section 5 far easier. We could have, and may yet, measure the system resources, such as CPU utilization, on the DUT, but we did not, because we were not concerned if the DUT was using 10 percent or 90 percent of its CPU, we only cared about how many packets it could forward.

As has been repeatedly pointed out over the last ten years taking full advantage of multi-core systems matters, and it will only matter more as core counts increase but CPU frequencies do not. The lesson of OpenBSD should not be ignored.

The addition of distributed systems coordination software, using Conductor has saved quite a bit of time and allowed us to run more experiments more frequently than we could have by hand. Most distributed systems testing is tedious setup work to get the sources, sinks and DUTs configured and then to collect the data. Having an automated system to aid in our efforts was a clear gain.

7 Status and Future Plans

Both the *netperf* and *Conductor* projects are current and ongoing. The relevant code repositories track both the work on automation as well as new test scenarios and results. It is our intention to continue to make periodic surveys of network performance on the BSDs in order to see how they change and improve over time.

8 Acknowledgments

This work has been directly supported by Rubicon Communications (Netgate) who has paid for the work involved to create the Conductor system as well as the time taken to compare and improve IPFW and PF in FreeBSD. We would especially like to thank Matthew Smith of Netgate for his help with the firewall testing. The FreeBSD Foundation and Sentex Communications support a test cluster that includes the high performance networking equipment used to carry out several of the tests described in this paper.

9 Bibliography

References

[1] Bryan Cantrill, Michael Shapiro, and Adam Leventhal. Dynamic Instrumentation of Production Systems. In *USENIX Annual Technical Conference*, page 137. ACM Request Permissions, June 2007.

[2] M. McKusick G. Neville-Neil, R. Watson. *The Design and Implementation of the FreeBSD Operating System.* Pearson Education, 2014.

[3] R. Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling.* Wiley-Interscience, 1991.

[4] Luigi Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *SIGCOMM Computer Communication Review*, 27(1):31–41, January 1997.

[5] Luigi Rizzo. netmap: A Novel Framework for Fast Packet I/O. In *Presented as part of the 2012 USENIX Annual Technical Conference*, pages 101–112, Boston, MA, 2012. USENIX.