The Anatomy of Networking in High-Frequency Trading

Peter P. Waskiewicz Jr. (PJ)
Jump Trading
Chicago, IL, USA
pwaskiewicz@jumptrading.com

Abstract

Networking has always served a number of very diverse environments. From Enterprise to the Cloud, Telco and edge, networking technologies have been able to use a "some sizes fit most" approach. This is good when it comes to supporting these technologies in the Linux kernel.

More specialized environments, such as High-Frequency Trading (HFT), have radically different networking requirements. Depending on the use case, one requirement might be that latency is paramount when interfacing with the market exchanges. Another use case might be in the HPC environment, where latency is still paramount, but sustained and reliable throughput is a must across grid networks.

This talk is intended to highlight where the Linux kernel networking stack intersects these requirements for HFT, and where it does not. It will also expand on how latency and jitter within HFT systems compare to "traditional" networking environments. Ultimately this talk is intended to generate discussion where HFT networking needs can help improve the existing intersection points in the kernel, and discuss where further native integration could be achieved.

Keywords

networking, kernel, xdp, offloads, trading, finance, performance

Introduction

Networking technologies have always been the most complex pieces of a modern computing system. Some technologies focus on general network connectivity, such as supporting a mobile device or personal computer, and some technologies focus on Software-Defined Networks, stitching together countless virtual machines and containers within data centers.

Modern operating systems, such as Linux, have evolved over many years to include very robust network stacks to serve these very diverse needs. Linux has found great success in adoption and datacenter footprint mainly due to the diversity and robustness of its network stack. It powers the majority of the largest Cloud Service Providers in the world, many Enterprise-level deployments, and is behind the largest install-base of handheld OS's in use today, Android.

Even with all of this diversity and flexibility, there are still environments where the Linux network stack sacrifices targeted performance for this flexibility. High-Frequency Trading environments, within the Finanical Tech world, have some fairly unique requirements for network behavior and performance.

This paper will focus on how HFT networking requirements match up against the diverse network stack capabilities of Linux. It will dive into:

- How synthentic benchmarking of traditional networks stack up to real-world benchmarks of traditional networks
- How the synthetic benchmarking resembles the beginning of HFT networking requirements in practice
- Analyzing what limitations the Linux network stack has compared to current HFT requirements for predictable latencies, and what might be done about it
- Exploring additional aspects of HFT networking requirements that encompass High-Performance Computing (HPC) grid environments, and how the Linux network stack compares to those requirements

"Traditional Networking", and the Synthetic Benchmarking Problem

Networking technology is one that fits into every modern workload in some fashion. In more "traditional" settings, such as datacenter-based networks (Enterprise), or disaggregated networks (Cloud), trying to nail down performance can be a difficult task. Very diverse environments create unpredictable workloads and interrupts that can introduce large amounts of jitter. When trying to benchmark these workloads, the benchmarking tests must be fairly synthetic to reduce or eliminate much of this jitter. But herein lies the issue; the benchmarks don't capture what happens in a production setting.

Benchmarking Setup

All benchmarks were captured via `netperf` [3] using the `TCP_RR` test. The system-under-test (SUT) is an Intel® Xeon® ES-2640 with a Broadcom bnx2x 10GbE adapter running stock

The views expressed in this paper are the author's only, and should not be attributed to Jump Trading.

Fedora 36. The peer system is an Intel® Xeon® Platinum 8180 with an Intel® 82599ES 10GbE adapter, running Gentoo Linux. The two systems are connected to a Mellanox SN2100 switch via Direct-Attach Twinax cables.

Note: this paper does not intend to serve as a reference on tuning systems for optimal scaling. Much of the techniques discussed in this paper have been used in many previous publications and presentations over the years. Two of those publications that this paper draws techniques from are here [1] and here [2].

Benchmark 1: Real-World Configuration

The simplest benchmark to run is one without any tuning. As one might expect, the results don't look terribly great.

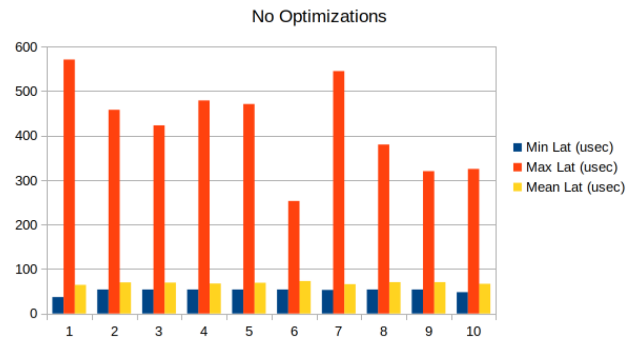

Figure 1: Netperf with no optimizations

As seen in Figure 1, one can see that every run produced drastically different latency figures. The average minimum latency is 51.6 µsecs, with an average mean latency of 68.7 µsecs. So not only is the jitter and variability significant, the latency numbers themselves are very high for an isolated test such as this.

The effects of no CPU affinity, no interrupt affinity, no cache isolation, all contribute to latency figures being rather poor. But the biggest issue is those latency figures are completely unpredictable, causing undeterministic workload behavior.

Benchmark 2: CPU affinity

One method to reduce jitter in a workload, and therefore increase the predictable performance of that workload, is to pin the workload to a particular CPU or set of CPUs. This allows data to stay cache-hot, as well as trying to keep memory buffers allocated from the local NUMA node.

In Figure 2, all of the tests were run while pinned to CPU 6 on NUMA node 0. This was chosen since the NIC itself is also in a PCIe slot that is on the PCIe tree attached to NUMA node 0 on this specific platform.

Overall jitter has improved slightly, and overall latency has also improved slightly. The average minimum latency across all tests is 50.1 µsecs, a 1.5 µsec delta. The average mean latency is 67.6 µsecs, another 1.1 µsec improvement over the previous benchmark. While these aren't significant improvements, they are still improvements that can have significant impact in the world of High-Frequency Trading. More on that later.

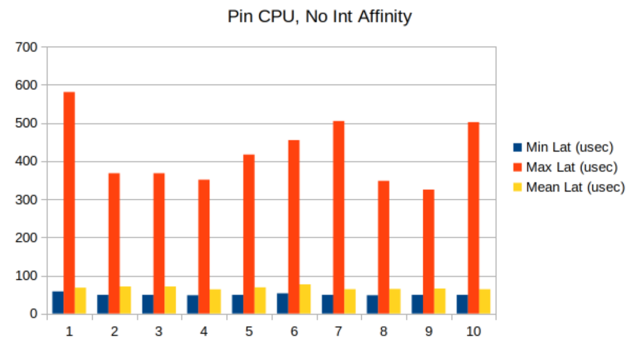

Figure 2: Netperf with CPU pinned

Benchmark 3: CPU affinity, Interrupt affinity

Taking the tuning tuning and benchmarking to the next level, the interrupt assigned to the corresponding NIC queue or queue pair can be pinned to a CPU. In the hope of being obvious, the CPU chosen should be the same one running the workload. In the example so far, that would be CPU 6. This also requires the `irqbalance` daemon to be stopped or killed, or use irqbalance policy scripts [4] to manually assign device interrupt affinity within the default balancing scheme.

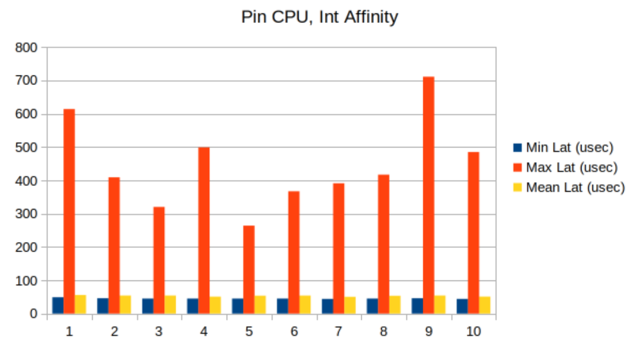

Figure 3: Netperf with CPU pinned, Interrupt pinned

Figure 3 illustrates even better latency and jitter improvements. With an average minimum latency of 45.4 µsecs and average mean latency of 53.1 µsecs, these are improvements of 6.2 µsecs and 15.6 µsecs, respectively.

As we're observing better predictable latency, we are also approaching configurations that are very artificial. Most workloads can't operate with this level of CPU and interrupt pinning, since other workloads typically run on the same platforms, and would be negatively impacted by these very specific tunings.

Benchmark 4: CPU affinity, Interrupt affinity, CPU isolation

One last OS-level optimization that can be done is isolation of the CPU or CPU's themselves. Isolating a CPU removes it from the OS scheduler's view, and keeps it parked at idle until a task is explicitly scheduled to run on that CPU. This can be done via the `taskset` command, or the CPU affinity parameter of `netperf`.

This isolation can be done by using the `isolcpus` kernel boot parameter, or it can be done via other runtime mechanisms. For this test, the `isolcpus` was used to completely isolate CPU 6, the same CPU that `netperf` has been pinned to in the previous two benchmarks.

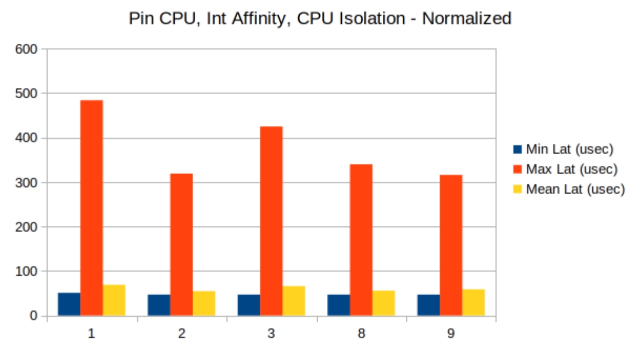

Figure 4: Netperf with CPU pinned, Interrupt pinned, CPU isolated

In Figure 4, the data here has been normalized. The maximum latencies in this dataset were drastically variable, making the min and average latencies difficult to analyze. In this dataset, the average minimum and average mean latencies are 47.8 µsecs and 60.9 µsecs. This is only a 3.8 µsecs and 7.8 µsecs improvement from the baseline, which is worse than without isolating the CPU.

A few factors come into play to explain this drop in predictable latency when the CPU is isolated. The biggest factor is the receive/Rx side of processing is done via `NAPI` in `SOFTIRQ` context. These context switches are always expensive. When they happen without the process scheduler involved, e.g. the CPU has been isolated, then more efficient overall use of the CPU is compromised. If improvement is desired, then one needs to look at tunables and changes outside of the kernel itself today.

Benchmark 5: CPU affinity, Interrupt affinity, CPU isolation, and Polling

When OS-level optimizations aren't sufficient to get complete predictable latencies, one must turn to application modifications. In this benchmark, some local changes were made to `netperf` to use local patches that polled for incoming Rx traffic. This also required changes to the `bnx2x` driver to not use interrupts, especially `NAPI`, to process incoming Rx traffic. The overall effect is the application, running on an isolated CPU core, never context switches to push or pull data from the NIC.

In Figure 5, our predictable latencies have tightened up very nicely, plus our overall latencies have dropped from our baseline. With an average min latency of 41.9 µsecs and average mean latency of 56.3 µsecs, these targets are 9.3 µsecs and 12.4 µsecs better than the baseline. With this specific platform, hardware, and switch, this is approaching the best case this setup is capable of.

These numbers, along with the remaining jitter, are very synthetic benchmarks that are not practical for traditional application and network workloads. However, the modifications, both OS and application, are typical examples for how High-Frequency Trading network environments operate in reality. And this is where the anatomy of these HFT-based networks begin to exist.

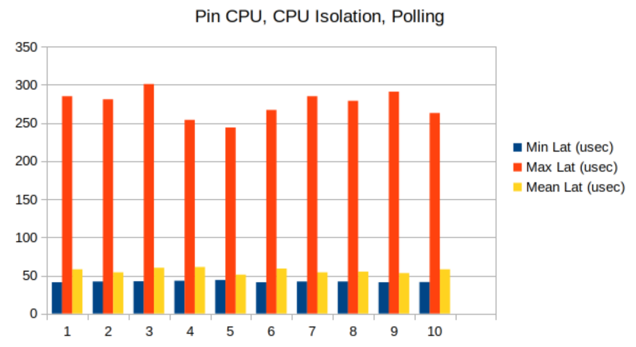

Figure 5: Netperf with CPU pinned, CPU isolated, Application polling

HFT Networking Basics: The Trading Puzzle

High-Frequency Trading firms and other related Financial Technology firms have notoriously been very secretive regarding their technologies. This includes usage of open-source software, or custom-built software frameworks where current OSS solutions fall short. This approach creates maintenance burdens, but also doesn't give back to OSS solutions where it can make sense to do so.

While custom solutions can close performance gaps and provide predictable latencies, these solutions still need to exist in a world where the financial exchanges use standard networking protocols. Exchanges such as the Chicago Mercantile Exchange (CME), NYSE, Eurex, and many others, all have their own protocols and data structure formats. But ultimately, they all communicate using standard Ethernet and other standard upper-layer protocols. They also have been mainly standardized on 10GbE networking. Because of this speed and feed standard, HFT networking typically doesn't care about the actual size of the bandwidth pipe available. The predictable latency and ability to eliminate jitter is paramount to the HFT networking model.

Trading Strategies vs. Latency

Within the financial world, trading strategies have always been present. The famous adage of "buy low, sell high" is one of the most obvious, and sound strategies. But how does one know when "low" is low, and "high" is high?

Trading strategies are the heart and soul of any financial model. Whether it's through broker accounts with individual-level trading, or HFT-based trading, the fact is the strategies are all data-driven. For HFT strategies, these can vary wildly what the strategy triggers on. Is it some market event, news headline, or some other mathematical and statistical pattern? Whatever the approach, data is crunched, and a trading decision is made. When this decision is executed, the trading decision itself must be executed with the financial exchange as soon as possible. The longer it takes for that decision to be acted upon decreases the likelihood that the circumstances leading to the decision will still be in play within the exchange. In other words: someone else will also figure out the same decision, and will try and execute their trade before you. If we scale this single event up to thousands of trades, or even hundreds of thousands, this latency between decision and execution is incredibly important. Taking it another step, the *predictable latency* is absolutely paramount.

Trading strategies assume that once a decision is made, it will be immediately executed. This translates to the elimination of as much latency in the network path as possible. From a Linux perspective, that does mean kernel bypass. Having software push data directly to hardware, and then hardware dumping data directly to userspace applications, is the most effective way to eliminate latency. But this also requires an environment that eliminates as much jitter as possible. Referring back to Figure 5, this synthetic environment with CPU isolation, CPU affinity, and application-awareness to poll for data, is exactly how the HFT environment needs to operate to achieve predictable latency.

Proposal: AF_XDP to the Rescue?

Kernel bypass has been frowned upon by kernel developers since the beginning of time. Why bypass a network stack that is hardened and validated in installations across the globe? But the realities of HFT networking and predictable latency drives that decision to bypass the networking stack.

`AF_XDP` [7] may be a solution to converge usage of the Linux network stack, and also provide the bypass semantics needed by HFT for predictable latency. If a device driver can be fully converted to a polling driver, meaning `NAPI` and `SOFTIRQ` are removed from the equation, then the context switching and interrupt-related jitter can be eliminated.

If such a polling driver existed, and a proper XDP program was loaded with an `AF_XDP`-attached socket, then hotpath data coming in from an exchange could easily be pushed to applications polling on incoming data. The huge advantage here though, is any traffic that is not critical to the hotpath and exchange traffic could be routed to the kernel, since it wouldn't trigger a hit in the `AF_XDP` application. Traffic such as LLDP or other control traffic would not need to be handled in custom ways, and could let the kernel itself handle it. This opens up many possibilities that can eliminate custom solutions, and gain benefits from using the native Linux network stack.

Why hasn't this transition been done yet? The main issue at the time of this writing is `AF_XDP` still has limitations with eliminating context switching on the Tx/transmit side. It still requires a `sendmsg()` in the simplest situations, which the system call itself would introduce latency and jitter through the context switch. This is still an area the kernel can improve and close this gap, providing kernel bypass with kernel stack integration.

HFT and HPC: Scaling Different Networks

High-Frequency Trading is driven by the quantitative research side that crunches massive amounts of data to derive trading strategies. In order to do this, a completely separate computing environment is typically required. These large High-Performance Computing, or HPC, environments, are typically a Grid network.

Grid networks typically operate on large datasets, potentially in the petabytes-range. While this amount of data could require very large data pipes, the effectiveness of the Grid network relies solely on the latency between nodes. If an application stalls waiting for results or extra data to crunch on, that damages the viability of the strategy. And if the network interconnect is to blame, then that latency is something that needs to be eliminated.

HPC and RDMA

RDMA, or Remote DMA, is a protocol and technique to try and eliminate latency on a network interconnect. RDMA networks also typically use Infiniband as their link-layer protocol, instead of Ethernet. This means these networks have dedicated switches and Infiniband adapters, and require different management, driver stacks, and software stacks, to run the RDMA and Message Passing Interface (MPI) data streams.

RDMA is a very popular choice in HPC and Grid networks, given its low and very predictable latencies. However, as Ethernet-based RDMA technologies continue to evolve and improve, such as RDMA over Converged Ethernet (RoCE), moving to these more standard link-layer networks is becoming more attractive.

HPC Without RDMA?

New techniques for passing data through the kernel, such as `io_uring` [6], are showing serious promise. `io_uring` started as a replacement for `libaio`, but has grown well beyond that initial focus area. The move towards using it for network traffic may upend the traditional BSD socket API. This could be a very strong potential for HPC networks to use instead of RDMA.

Another promising protocol is Homa [5]. It is meant to replace other datacenter protocols, and promising very low latency. An interesting approach though is that Homa can be used outside the kernel with kernel bypass, or as a protocol within the kernel stack. This is still fairly early in development, but shows great potential to be a candidate for HPC and HFT networks alike.

Conclusion

High-Frequency Trading has very demanding and unique requirements for networking. On the trading side, predictable latency is key for the success of trading algorithms and strategies. Without the predictable latency, these strategies will fail, negatively impacting both the Trading Firm, but also the markets themselves. But the custom nature of many things in the HFT space has become more complex to maintain, and any opportunity that the kernel can provide to ease that custom maintenance burden is very welcome. "Kernel-bypass-with-the-kernel" with `AF_XDP` is a very promising operating model. Eliminating the context switches, and then providing kernel-managed polling models, will further narrow the gaps.

In the HPC space, RDMA-based technologies can continue converging to more common protocol and hardware technologies. Promising technologies such as `io_uring` [6], and new protocols such as Homa [5], continue to narrow the gap between highly customized networks, and something that the kernel can provide out of the box.

As these technologies continue to move forward, the HFT community needs to continue engaging with the Linux kernel community to help guide this path to predictable latency on any workload.

Acknowledgments

I would like to acknowledge the NetDev 0x16 program committee for the opportunity to submit and invitation to present this paper.

References

[1] Luca Deri, Joseph Gasparakis, PJ Waskiewicz, Francesco Fusco 2010. *Wire-Speed Hardware-Assisted Traffic Filtering with Mainstream Network Adapters* NEMA, Proceedings from First International Workshop on Network Embedded Management and Applications

[2] Luca Deri, Francesco Fusco 2010. *High Speed Network Traffic Analysis with Commodity Multi-core Systems* https://luca.ntop.org/imc2010.pdf

[3] Rick Jones, et al 2008. *Netperf main github site* https://github.com/HewlettPackard

[4] Neil Horman, PJ Waskiewicz, et al 2012. *irqbalance main github site* https://github.com/Irqbalance/irqbalance

[5] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, John Ousterhout 2018. *Homa: A Receiver-Driven Low-Latency Transport Protocol Using Network Priorities* SIGCOMM, Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication Budapest, Hungary

[6] Jens Axboe 2019. *Efficient IO with io_ring* https://kernel.dk/io_uring.pdf

[7] Jonathan Corbet, Björn Töpel 2018. *Accelerating networking with AF_XDP* https://lwn.net/Articles/750845/

Author Biography

Peter Waskiewicz Jr (PJ) is a Senior Software Engineer in Jump Trading's core engineering division, focusing on Linux kernel and device driver development, along with other system-level engineering. Prior to Jump Trading, PJ spent the majority of his career at Intel, where he was responsible for writing and maintaining several of the Intel Ethernet Linux device drivers, and developing Linux kernel changes for scaling to 10GbE and beyond. PJ was also a Senior Principal Engineer at NetApp in the SolidFire division, where he was the chief Linux kernel and networking architect for the SolidFire scale-out cloud storage platform. He is also an adjunct faculty at Portland State University, teaching OS and Device Drivers in the Electrical and Computer Engineering Department.