Decompressing TCP/IP Headers for High-Speed Links (or How to Build a 100Gb/s Network Stack)

Under submission for review.

ABSTRACT

Network protocol implementations are stuck in the 1980s. Bits and bytes are squeezed next to each other to maximise efficiency for slow link speeds and fast CPUs. However, CPU speeds have plateaued for several years and network speeds have increased rapidly. Single CPU cores can no longer hope to keep up with doing per-packet network encoding and decoding.

We investigate several options for expanding, aligning, and sizing TCP/IP[1] and Ethernet packets to maximise efficiency for our (now) slow, 64-bit, little-endian, CPUs. In these first steps toward a practical, high speed network stack, we show that combining Direct Cache Access (DCA)[2] with real zero-copy delivery and restructured 64-bit alignment of packet fields are crucial to packet processing beyond 40Gb/s. Further, we demonstrate a simple user-space network stack capable of handling over 100Gb/s of traffic on a single core and maintaining up to 100Gb/s per core over multiple cores.

1. INTRODUCTION

By classical wisdom, smaller packet headers are better: fewer bits wasted on the wire mean greater efficiency overall. Van Jacobson's impressive RFC 1144 [2] "*Compressing TCP/IP Headers for Low-Speed Serial Links*" takes this concept to the extreme. It shows how to compress the 40 byte TCP/IP header down to less than 4 bytes in the common case.

Despite the heroic efforts of Jacobson and others to keep TCP/IP headers short, saving bits on the wire is only beneficial if the cost of transmission outweighs the cost of compression. For example, on a 9600b/s serial link, each bit costs around 10 microseconds to transmit. By saving 36 bytes per packet, Jacobson could save just under 30 milliseconds per transmission. From a total round trip budget of 100 milliseconds, 30 milliseconds represented an impressive saving.

On a modern 40Gb/s Ethernet link, each bit now costs 25 picoseconds to transmit. A best-in-class CPU running at 4GHz executes only one cycle every 250 picoseconds. This means that the CPU needs to be able to process an average of 10 bits per cycle to keep up with line-rate. As we go forward, power and thermal efficiency issues are causing CPUs to get slower and wider rather than faster. Intel's new Haswell CPUs typically execute 1 cycle every 500 picoseconds (2GHz). By contrast, networks are getting faster. A 100Gb/s Ethernet link can transmit 1 bit every 10 picoseconds, a factor of 50x faster. This means that CPUs now need to process on average 50 bits per cycle just to keep up.

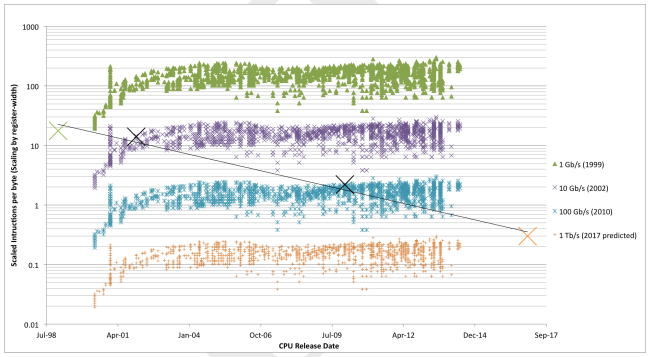

Figure 1: Instructions per network-byte for a range of Ethernet technologies. Despite scaling by instruction width, the flattening of CPU instruction-rate is clear.

A 64-bit CPU should be able to cope with 64 bits per cycle, as long as each instruction requires only one cycle. Modern CPUs typically require multiple cycles for many instructions. By packing bits tightly together and by aligning fields on 16-bit (Ethernet) or 32-bit (TCP/IP) boundaries, we guarantee that the CPU will need to do more than 1 instruction per header field.

It gets worse. Network packets are transmitted using network byte ordering which is "big-endian", but the majority of CPUs today use "little-endian" byte ordering internally. Once the CPU has found and extracted the correct field from a packet, it then needs to reorder the bytes into little-endian format. The little-endian conversion adds strain to an already tight cycle budget.

From these examples, it should be clear that CPU cycles are becoming a rare commodity and that network bits are now plentiful. We think that the time has come to reevaluate the tradeoffs in on-the-wire formats for TCP/IP protocols. Specifically, we consider if there are optimisations to the layout of packets in host memory only, without changing

[1] **Malte: We don't really do TCP...**
[2] **Malte: Do we have any numbers on this?**

the functionality of the protocols themselves. To do this, we build a simple network stack that runs directly out of host memory. We then test a variety of different packet layouts and show that gains of nearly 3× can be found by arranging packet wire formats correctly, allowing network stacks to scale beyond 40Gb/s and up to over 100Gb/s.

```
foreach packet in packet_buffer do
    start ← timer;
    packet.ethernet.src ← 0xFFFFFF;
    packet.ethernet.dst ← 0x000001;
    packet.ethernet.type ← 0x0800;
    ...
    packet.ip.size ← big_endian(IP_SIZE);
    ...
    packet.ip.protocol ← 0x11;
    ...
    packet.udp.size ← big_endian(UDP_SIZE);
    ...
    for i ← 0 to UDP_SIZE ÷ WORD_SIZE do
        packet.udp.data[i] ← const integer;
    end
    stop ← timer;
end
```

Algorithm 1: Generating packets

2. TESTING HIGH SPEED NETWORKS

To test the effect of packet layouts on processing speed, we would like to use a flexible, high speed network adapter. Some FPGA based 100Gb/s network adapters are beginning to appear on the market [1], but they are rare and expensive. Additionally, adapting 1000's of lines of legacy network stack code to process new packet formats will be time consuming. To work around these problems, we take a different approach. Instead of building a network stack for a particular adapter, we use our experience in high speed network adapter design *[reference removed for blind review]* to build a generic network stack that runs directly out of system memory. We assume that an adapter will eventually become available to place packets there. This allows us to test the absolute speed of packet processing quickly and flexibly It also gives us an indication of the expected upper limits that software could expect to achieve. By writing our own network stack, we free ourselves from legacy code bases and can ensure that our packet processor is flexible enough to cope with a range of different packet formats.

Our initial network stack only processes UDP packets, delivered over IP version 4.0 transport using Ethernet frames. In principle any protocol or transport could be supported, but this is the simplest to implement. By doing so, we can get a strong indication of the practical upper bounds of any reasonable network stack.

Our network stack is divided into two parts: generating and receiving packets. Packet generation is detailed in Algorithm 1 and packet receiving is detailed in Algorithm 2. For each experiment, we first run the generation phase and then the receiving phase. This way we can test both parts independently. As with any sensible network stack, we only perform endian conversions for values that are non-constant. For example, the ethernet packet type field is a constant, but the IP header size is not. Our packet buffer is pinned so that swapping cannot happen and aligned to a 4kB page boundary.

Our experiments are run using a top-of-the range Intel Core i7 Ivy Bridge CPU (4930K), with 12 threads running at 3.4GHz. We ensure that the network-stack process is pinned to a unique CPU, with no other processes or hyperthreads interfering and that the process is run with full-realtime priority. This environment is overly generous. Real in-kernel network stacks have limited time to run and have to contend with interrupts and threads from other competing resources.

```
foreach packet in packet_buffer do
    start ← timer;
    if packet.ethernet.src ≠ 0xFFFFFF then
        continue;
    end
    if packet.ethernet.dst ≠ 0x000001 then
        continue;
    end
    if packet.ethernet.type ≠ 0x0800 then
        continue;
    end
    ...
    ip_size ← little_endian(packet.ip.size);
    total_ip_bytes ← total_ip_bytes + ip_size;
    ...
    if packet.ip.protocol ≠ 0x11 then
        continue;
    end
    ...
    udp_size ← little_endian(packet.udp.size);
    total_udp_bytes ← total_udp_bytes + udp_size;
    ...
    counter ← 0;
    for i ← 0 to udp_size ÷ WORD_SIZE do
        counter ← packet.udp.data[i];
    end
    stop ← timer;
end
```

Algorithm 2: Receiving packets

3. EVALUATION

3.1 Baseline

Discuss the current performance speeds with packet layouts how they are.

3.2 Memory speed

Show how zero copy and memory allocation (eg DCA) affect the overall performance.

3.3 Network Byte Ordering

Show how network byte ordering affects the speed.

3.4 Packet layouts

Show how network byte ordering affects the speed.

3.5 Core Scaling

Show how this scales as we increase core counts.

4. DISCUSSION

5. RELATED WORK

ToDo: AWM comments:

1. *isn't this all solved by hardware offload stuff?* – not sure what the answer is there[3]
2. *but my machine has lots of cores...* – not helpful, since packet decoding is unsplittable and cannot be parallelised. Even a single TCP flow cannot be parallelised over multiple cores.

The penetration of hardware offload features promotes a belief that we have solved the mismatch between network speed and host capacity. However, beyond the most rudimentary offload — those plausible on a packet-by-packet basis: checksum offload, packetization and packet aggregation — offload has not proven to be widely useful.[4] Mogul [3] provided a sober insight into this by noting that best-use offload was in extremely specialised circumstance: the per-application networking including per-application offload engines. However, in opposition to this networking continues to be the provenance of the operating system: providing a generic service to every application. Thus providing limited ability to specialise tasks for specific hardware. [5]

6. CONCLUSIONS

References

[1] INVEATECH. COMBO-100G FPGA Card, July 2014.

[2] JACOBSON, V. Compressing TCP/IP Headers for Low-Speed Serial Links. RFC 1144 (Proposed Standard), Feb. 1990.

[3] MOGUL, J. C. Tcp offload is a dumb idea whose time has come. In *Proceedings of the 9th Conference on Hot Topics in Operating Systems - Volume 9* (2003), HOTOS'03, pp. 25–30.

[3] **Malte: a NIC could potentially DMA larger, better-aligned frames, but that would be wasteful of bus bandwidht...**

[4] **Malte: why?**

[5] **AWM: jibberish written by a tired brain**