Improving Driver Performance – A Worked Example

Bruce Richardson
Legal Disclaimers

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

This document contains information on products, services and/or processes in development. All information provided here is subject to change without notice. Contact your Intel representative to obtain the latest forecast, schedule, specifications and roadmaps.

Intel technologies’ features and benefits depend on system configuration and may require enabled hardware, software or service activation. Performance varies depending on system configuration. No computer system can be absolutely secure. Check with your system manufacturer or retailer or learn more at intel.com.

© 2016 Intel Corporation. Intel, the Intel logo, Intel. Experience What’s Inside, and the Intel. Experience What’s Inside logo are trademarks of Intel Corporation in the U.S. and/or other countries.

*Other names and brands may be claimed as the property of others.
Introduction

• This is a real-world use-case of performance improvement using performance analysis tools – in this case “Intel® VTune™ Amplifier”

• The issues discovered here were largely discovered by accident when investigating other DPDK behaviour

• Hope the walk-through of the issue debug may be of use to others when dealing with performance issues
Intel® Vtune Amplifier

- Tool for performance analysis and debugging
- Uses hardware event counters to report on issues affecting program execution, e.g. CPU stalls due to memory access
Test Setup

Traffic Generator

2 x Intel® Ethernet Converged Network Adapter XL710-QDA1

DUT (Intel® Xeon® E5-2699 v3 @ 2.30GHz)
**L1 Bound**:

This metric shows how often machine was stalled without missing the L1 data cache. The L1 cache typically has the shortest latency. However, in certain cases like loads blocked on older stores, a load might suffer a high latency even though it is being satisfied by the L1. Note that this metric value may be highlighted due to DTLB Overhead or Cycles of 1 Port Utilized issues.

- **DTLB Overhead**:
  
- **Loads Blocked by Store Forwarding**:

Loads are blocked during store forwarding when the forwarding store is smaller than the load, change the store, or if a load misses in the forwarding store cache. Use source/assembly view to identify the blocked loads, then identify the problematically-forwarded loads. To streamline memory operations in the pipeline, a load can avoid waiting for memory if a prior store, still in flight, is writing the data that the load wants to read (a ‘store forwarding’ process). However, in some cases, generally when the prior store is writing a smaller region than the load is reading, the load is blocked for a significant time pending the store forward. This metric measures the performance penalty of such blocked loads.

**L2 Bound**:

This metric does a rough estimation of a blocked load on the L2 cache.

**L3 Bound**:

This metric shows how often CPU was stalled on L3 cache, or contended with a sibling Core. Avoiding cache misses (L2 misses/L3 hits) improves the latency and increases performance.
What does this mean?

- We are doing a write to a memory location
- We are subsequently doing a read from that memory location
- The read is getting blocked by the write because the read is for more data than just the write
  - if the write is for the same amount of data, then we can do “store forwarding” to return the data write to the read without hitting cache/mem
  - if the read can be delayed till later, the write can complete and the read can come from cache
- This should not generally show up as a problem in good code
/* D.1 pkt 3,4 convert format from desc to pktmbuf */
pkt_mb4 = _mm_shuffle_epi8(descs[3], shuf_msk);
pkt_mb3 = _mm_shuffle_epi8(descs[2], shuf_msk);

vmovdqa8 -0x18(%rsp), %xmm11
vmovdqa8 -0x28(%rsp), %xmm1
vmovdqa8 -0x48(%rsp), %xmm3
Reading this case

• In this case, both the instruction highlighted and the previous one are 128-bit loads.
• Therefore either one is a potential candidate for the source of the delay
• If we assume that this is the read, then we need to find the offending write:
  • occurs previous to these [nice and easy to find in C, as there is a compiler barrier]
  • is of a size smaller than 128-bits
Other Weirdness…

• Why does the code:
  \[
  \text{pkt_mb3} = \_\text{mm\_shuffle\_epi8(descs[2], shuf\_msk)};
  \]
  cause a read from memory at all?

• Shouldn’t descs[2] have already been loaded to xmm register previously at the line?
  \[
  \text{descs[2]} = \_\text{mm\_loadu\_si128((\_m128i *) (rxdp + 2))};
  \]

• Let’s look at that previous load lines in vtune…
descs[3] = _mm_loadu_si128((__m128i *)(rxdp + 3))

vmovdqux 0x60(%r9), %xmm3
vmovapsx %xmm3, -0x18(%rsp)
What is happening?

• A load intrinsic is resulting in an xmm load followed by a store?
• We have a second mystery.
• Let’s trace back through what happens to the descriptors through the code between the two points…
desc_pktlen_align

- Only work done between desc[2] load and the offending line is function “desc_pktlen_align”
- Again look at assembler listing
Vector Code

[Including writes]

Scalar Code
When assigning the lengths, we use 16-bit writes:
- have to go to memory (can’t assign to an xmm register)
- causes the xmm load to immediately store to stack
- causes the shuffle op to trigger a second load
- That second load (128b) blocks on 16b write

```
pktlen0 = _mm_srli_epi32(pktlen0, PKTLEN_SHIFT);
pktlen0 = _mm_and_si128(pktlen0, pktlen_msk);
pktlen0 = _mm_packs_epi32(pktlen0, zero);
vol.dword = _mm_cvtsi128_si64(pktlen0);

/* let the descriptor byte 15-14 store the pkt len */
*((uint16_t *)&descs[0]+7) = vol.e[0];
*((uint16_t *)&descs[1]+7) = vol.e[1];
*((uint16_t *)&descs[2]+7) = vol.e[2];
*((uint16_t *)&descs[3]+7) = vol.e[3];
```
The Fix

Rewrite to use entirely vector operations:

• keeps things in xmm registers
• saves unnecessary loads and stores
• prevents store forward errors
• improves performance.
• makes people happy*

*NOTE: happiness not guaranteed
Result

- Stalls due to Loads Blocked by Store Forwarding dropped about 19x:
  - Before: 0.189
  - After: 0.010

- Overall PMD performance measured by testpmd increased by over 5%.