DPDK

USERSPACE, October 2016

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.

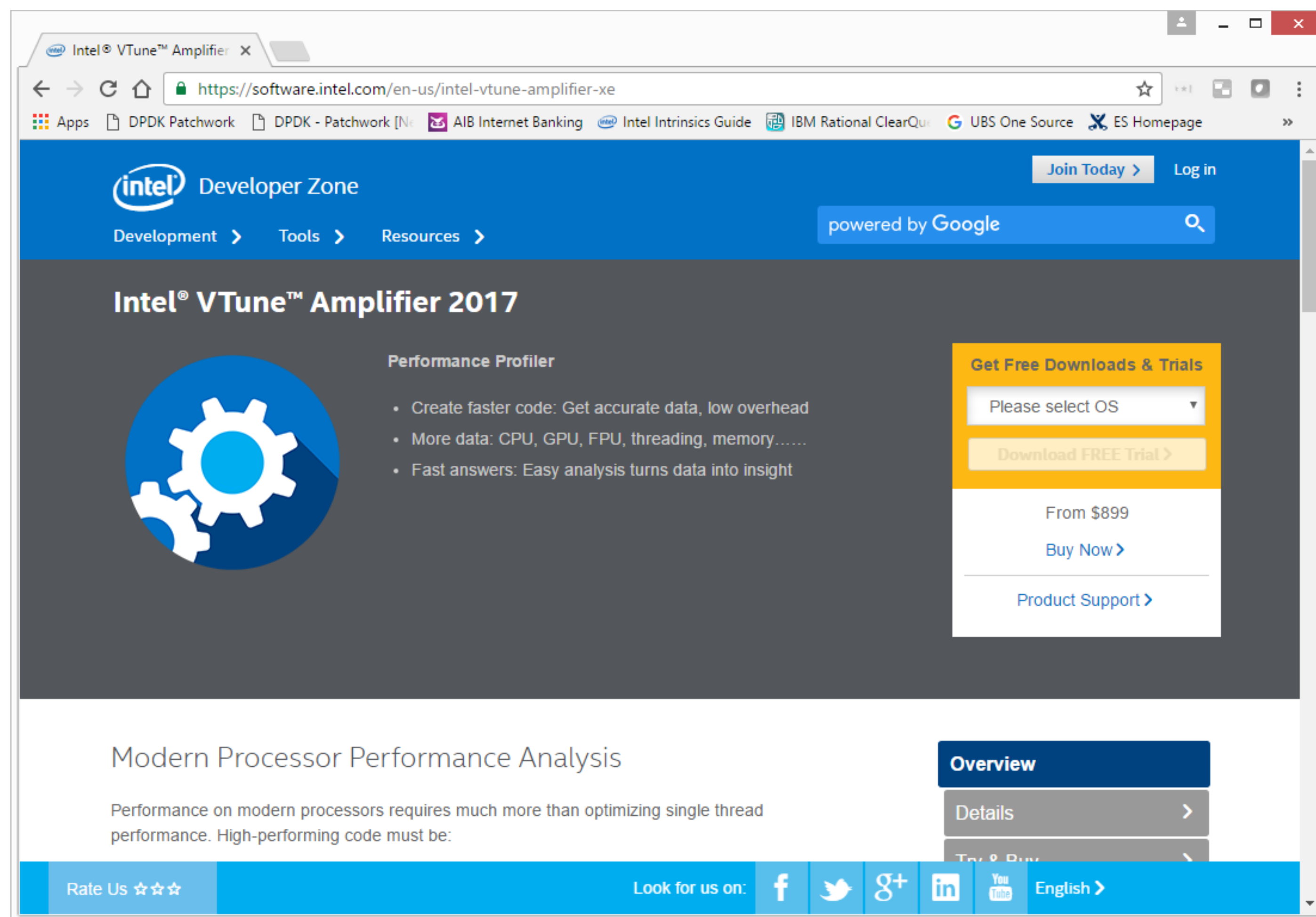
DPDK
DATA PLANE DEVELOPMENT KIT

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

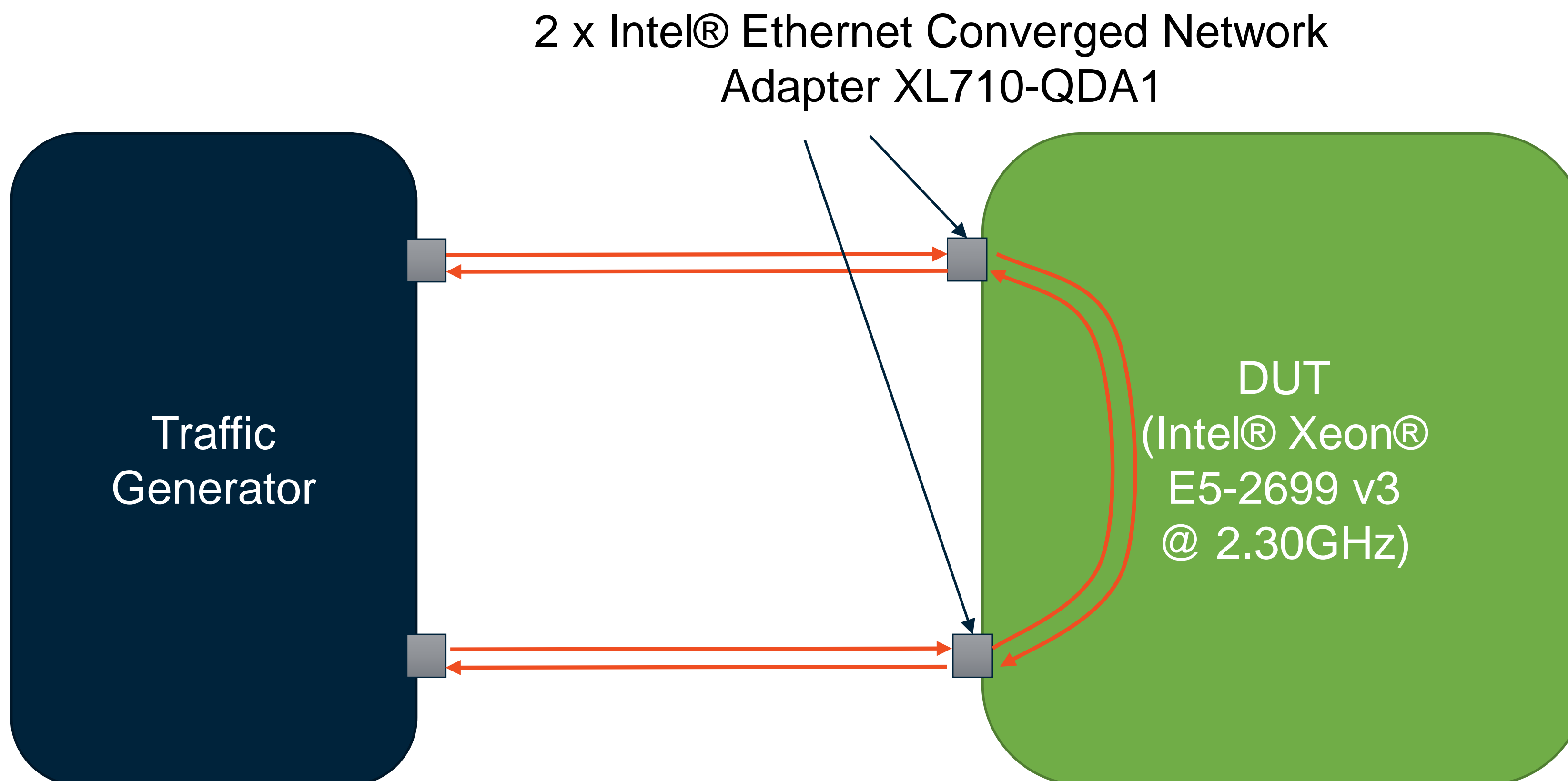
Intel® Vtune Amplifier

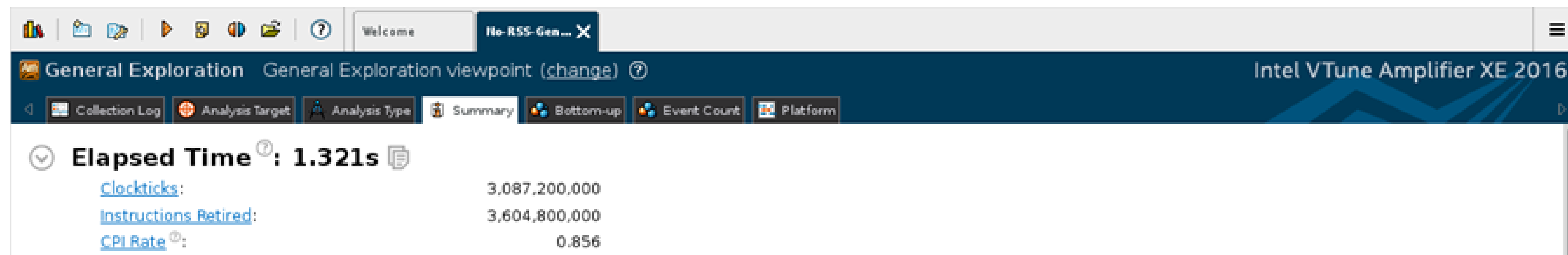
- Tool for performance analysis and debugging
- Uses hardware event counters to report on issues affecting program execution, e.g. CPU stalls due to memory access



DPDK
DATA PLANE DEVELOPMENT KIT

Test Setup





L1 Bound: **0.288**

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: 0.000

Loads Blocked by Store Forwarding: 0.189

Loads are blocked during store for[...]es. Use source/assembly view to identify the blocked loads, then identify the problematically-forwarde[...]en dynamic instructions prior to the load. If the forwarding store is smaller than the load, change the st[...]

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 signficant time pending the store forward. This metric measures the performance penalty of such blocked loads.

Split Loads:

4K Aliasing:

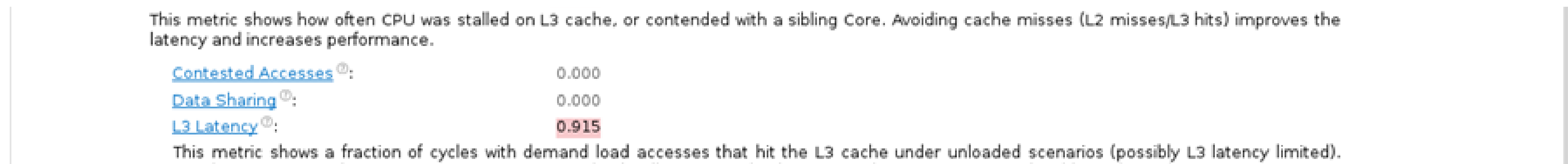
FB Full:

This metric does a rough estimation [...]ted additional demand L1D demand requests to proceed.

L2 Bound: 0.000

L3 Bound: **0.179**

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.



DPDK
DATA PLANE DEVELOPMENT KIT

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

DPDK
DATA PLANE DEVELOPMENT KIT

General Exploration General Exploration viewpoint (change)

Intel VTune Amplifier XE 2016

Analysis Target | Analysis Type | Collection Log | Summary | Bottom-up | Event Count | Platform

Source | Assembly | Assembly grouping: Address

```
326  /* 8.1 load 2 mbuf point */
327  descs[1] = _mm_loadu_si128((__m128i *)(rxdp + 1));
328  descs[0] = _mm_loadu_si128((__m128i *)(rxdp));
329
330  /* 8.2 copy 2 mbuf point into rx_pkts */
331  _mm_storeu_si128((__m128i *)&rx_pkts[pos+2], mbp2);
332
333  if (split_packet) {
334      rte_prefetch0(&rx_pkts[pos]->cacheline1);
335      rte_prefetch0(&rx_pkts[pos + 1]->cacheline1);
336      rte_prefetch0(&rx_pkts[pos + 2]->cacheline1);
337      rte_prefetch0(&rx_pkts[pos + 3]->cacheline1);
338  }
339
340  /*shift the pktlen field*/
341  desc_pktlen_align(descs);
342
343  /* avoid compiler reorder optimization */
344  rte_compiler_barrier();
345
346  /* D.1 pkt 3,4 convert format from desc to pktmbuf */
347  pkt_mb4 = _mm_shuffle_epi8(descs[3], shuf_msk);
348  pkt_mb3 = _mm_shuffle_epi8(descs[2], shuf_msk);
...
363  pkt_mb1 = _mm_shuffle_epi8(descs[0], shuf_msk);
364
365  /* C.2 get 4 pkts staterr value */
366  zero = _mm_xor_si128(dd_check, dd_check);
367  staterr = _mm_unpacklo_epi32(sterr_tmp1, sterr_tmp2);
368
369  /* D.3 copy final 3,4 data to rx_pkts */
370  _mm_storeu_si128((void *)&rx_pkts[pos+3]->rx_descriptor_fields1,
371          pkt_mb4);
```

```
0x52d49f        Block 6:
0x52d49f  341   vpunpckhdq %xmm1, %xmm0, %xmm1
0x52d4a3  341   vpunpckhdq %xmm3, %xmm2, %xmm0
0x52d4a7  341   vpunpckhdq %xmm0, %xmm1, %xmm0
0x52d4ab  341   vpsrld $0x6, %xmm0, %xmm0
0x52d4b0  341   vpand %xmm0, %xmm15, %xmm0
0x52d4b4  341   vpackssdw %xmm8, %xmm0, %xmm0
0x52d4b9  341   vmovq %xmm0, %rdx
0x52d4be  341   mov %rdx, %r10
0x52d4c1  341   movw %dx, -0x3a(%rsp)
0x52d4c6  341   shr $0x10, %r10
0x52d4ca  341   movw %r10w, -0x2a(%rsp)
0x52d4d0  341   mov %rdx, %r10
0x52d4d3  341   shr $0x30, %rdx
0x52d4d7  341   shr $0x20, %r10
0x52d4db  341   movw %dx, -0xa(%rsp)
0x52d4e0  341   movw %r10w, -0x1a(%rsp)
0x52d4e6  347   vmovdqax -0x18(%rsp), %xmm11
0x52d4ec  348   vmovdqax -0x28(%rsp), %xmm1
0x52d4f2        vmovdqax -0x48(%rsp), %xmm3
0x52d4fd  370   vpaddw %xmm6, %xmm7, %xmm6
0x52d501  353   vmovdqax -0x38(%rsp), %xmm4
0x52d507  348   vpshufb %xmm9, %xmm1, %xmm5
0x52d50c        vpunpckhdq %xmm1, %xmm11, %xmm0
0x52d540  355   vpshufb %xmm2, %xmm12, %xmm2
0x52d545  355   vpor %xmm2, %xmm1, %xmm2
0x52d549  355   vmovq %xmm2, %rdx
0x52d54e  363   vpshufb %xmm9, %xmm3, %xmm3
0x52d553  376   vpaddw %xmm4, %xmm7, %xmm4
0x52d557  355   movzx %dx, %ebp
0x52d55a  377   vpaddw %xmm3, %xmm7, %xmm3
0x52d55e  355   movq %rbp, 0x18(%r10)
0x52d562  355   movq (%rsi), %rbp
0x52d565  355   mov %rdx, %r10
```

```
/* 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);
```

```
vmovdqax -0x18(%rsp), %xmm11
vmovdqax -0x28(%rsp), %xmm1
vmovdqax -0x48(%rsp), %xmm3
```


DPDK
DATA PLANE DEVELOPMENT KIT

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

DPDK
DATA PLANE DEVELOPMENT KIT

Other Weirdness…

- Why does the code:

```
pkt_mb3 = _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?

```
descs[2] = _mm_loadu_si128((__m128i *)(rxdp + 2));
```

- Let’s look at that previous load lines in vtune…

```
311        __m128i mbp1, mbp2; /* two mbuf pointer in one XMM reg. */
312
313        /* B.1 load 1 mbuf point */
314        mbp1 = _mm_loadu_si128((__m128i *)&sw_ring[pos]);
315        /* Read desc statuses backwards to avoid race condition */
316        /* A.1 load 4 pkts desc */
317        descs[3] = _mm_loadu_si128((__m128i *)(rxdp + 3));
318
319        /* B.2 copy 2 mbuf point into rx_pkts  */
320        _mm_storeu_si128((__m128i *)&rx_pkts[pos], mbp1);
321
322        /* B.1 load 1 mbuf point */
323        mbp2 = _mm_loadu_si128((__m128i *)&sw_ring[pos+2]);
324
325        descs[2] = _mm_loadu_si128((__m128i *)(rxdp + 2));
326        /* B.1 load 2 mbuf point */
```

```
0x52d41c   305   xor %eax, %eax
0x52d41e   305   vmovdqax  0xd582a(%rip), %xmm12
0x52d426   240   vpinsrw $0x1, %eax, %xmm0, %xmm0
0x52d42b   240   vpunpckldq %xmm0, %xmm8, %xmm7
0x52d42f   240   vpunpckldq %xmm8, %xmm0, %xmm0
0x52d434   240   vpunpcklqdq %xmm0, %xmm7, %xmm7
0x52d438         Block 4:
0x52d438   314   vmovdqux  (%r11), %xmm0
0x52d43d   317   vmovdqux  0x60(%r9), %xmm3
0x52d443   317   vmovapsx  %xmm3, -0x18(%rsp)
0x52d449   320   vmovupsx  %xmm0, -0x8(%rsi)
0x52d44e   325   vmovdqux  0x40(%r9), %xmm1
0x52d454   323   vmovdqux  0x10(%r11), %xmm4
0x52d45a   325   vmovapsx  %xmm1, -0x28(%rsp)
0x52d460   327   vmovdqux  0x20(%r9), %xmm2
0x52d466   327   vmovapsx  %xmm2, -0x38(%rsp)
```

```
descs[3] = _mm_loadu_si128((__m128i *)(rxdp + 3))
```

```
vmovdqux 0x60(%r9), %xmm3
vmovapsx %xmm3, -0x18(%rsp)
```


DPDK
DATA PLANE DEVELOPMENT KIT

What is happening?

- A load instrinsic 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

DPDK
DATA PLANE DEVELOPMENT KIT

General Exploration — General Exploration viewpoint (change)

Intel VTune Amplifier XE 2016

Analysis Target | Analysis Type | Collection Log | Summary | Bottom-up | Event Count | Platform

Source | Assembly | Assembly grouping: Address

```
302      * D. fill info. from desc to mbuf
303      */
304
305     for (pos = 0, nb_pkts_recd = 0; pos < RTE_I40E_VPMD_RX_BURST;
306             pos += RTE_I40E_DESCS_PER_LOOP,
307             rxdp += RTE_I40E_DESCS_PER_LOOP) {
308         __m128i descs[RTE_I40E_DESCS_PER_LOOP];
309         __m128i pkt_mb1, pkt_mb2, pkt_mb3, pkt_mb4;
310         __m128i zero, staterr, sterr_tmp1, sterr_tmp2;
311         __m128i mbp1, mbp2; /* two mbuf pointer in one XMM reg. */
312
313         /* B.1 load 1 mbuf point */
314         mbp1 = _mm_loadu_si128((__m128i *)&sw_ring[pos]);
315         /* Read desc statuses backwards to avoid race condition */
316         /* A.1 load 4 pkts desc */
317         descs[3] = _mm_loadu_si128((__m128i *)(rxdp + 3));
318
319         /* B.2 copy 2 mbuf point into rx_pkts */
320         _mm_storeu_si128((__m128i *)&rx_pkts[pos], mbp1);
321
322         /* B.1 load 1 mbuf point */
323         mbp2 = _mm_loadu_si128((__m128i *)&sw_ring[pos+2]);
324
325         descs[2] = _mm_loadu_si128((__m128i *)(rxdp + 2));
326         /* B.1 load 2 mbuf point */
327         descs[1] = _mm_loadu_si128((__m128i *)(rxdp + 1));
328         descs[0] = _mm_loadu_si128((__m128i *)(rxdp));
329
330         /* B.2 copy 2 mbuf point into rx_pkts */
331         _mm_storeu_si128((__m128i *)&rx_pkts[pos+2], mbp2);
332
333         if (split_packet) {
334             rte_prefetch0(&rx_pkts[pos]->cacheline1);
335             rte_prefetch0(&rx_pkts[pos + 1]->cacheline1);
336             rte_prefetch0(&rx_pkts[pos + 2]->cacheline1);
337             rte_prefetch0(&rx_pkts[pos + 3]->cacheline1);
338         }
339
340         /*shift the pktlen field*/
341         desc_pktlen_align(descs);
342
343         /* avoid compiler reorder optimization */
344         rte_compiler_barrier();
345
346         /* D.1 pkt 3,4 convert format from desc to pktmbuf */
347         pkt_mb4 = _mm_shuffle_epi8(descs[3], shuf_msk);
```

Selected 1 row(s):

```
Address    Source Line  Assembly
0x52d489   335   movq (%rsi), %rdx
0x52d48c   335   prefetcht0z 0x40(%rdx)
0x52d490   336   movq 0x8(%rsi), %rdx
0x52d494   336   prefetcht0z 0x40(%rdx)
0x52d498   337   movq (%r8), %rdx
0x52d49b   337   prefetcht0z 0x40(%rdx)
0x52d49f
0x52d49f   341   vpunpckhdq %xmm1, %xmm0, %xmm1
0x52d4a3   341   vpunpckhdq %xmm3, %xmm2, %xmm0
0x52d4a7   341   vpunpckhdq %xmm0, %xmm1, %xmm0
0x52d4ab   341   vpsrld $0x6, %xmm0, %xmm0
0x52d4b0   341   vpand %xmm0, %xmm15, %xmm0
0x52d4b4   341   vpackssdw %xmm8, %xmm0, %xmm0
0x52d4b9   341
0x52d4be   341
0x52d4c1   341   movw %dx, -0x3a(%rsp)
0x52d4c6   341   shr $0x10, %r10
0x52d4ca   341   movw %r10w, -0x2a(%rsp)
0x52d4d0   341   mov %rdx, %r10
0x52d4d3   341   shr $0x30, %rdx
0x52d4d7   341   shr $0x20, %r10
0x52d4db   341   movw %dx, -0xa(%rsp)
0x52d4e0   341   movw %r10w, -0x1a(%rsp)
0x52d4e6   347   vmovdqax -0x18(%rsp), %xmm11
0x52d4ec   348   vmovdqax -0x28(%rsp), %xmm1
0x52d4f2   353   vmovdqax -0x48(%rsp), %xmm3
0x52d4f8   347   vpshufb %xmm9, %xmm11, %xmm6
0x52d4fd   370   vpaddw %xmm6, %xmm7, %xmm6
0x52d501   353   vmovdqax -0x38(%rsp), %xmm4
0x52d507   348   vpshufb %xmm9, %xmm1, %xmm5
0x52d50c   351   vpunpckhdq %xmm1, %xmm11, %xmm0
0x52d510   355   vpunpckhwd %xmm11, %xmm1, %xmm1
0x52d515   355   movq -0x8(%rsi), %r10
0x52d519   372   vpaddw %xmm5, %xmm7, %xmm5
0x52d51d   355   vpunpckhwd %xmm4, %xmm3, %xmm2
0x52d521   355   vpunpckldq %xmm1, %xmm2, %xmm2
0x52d525   355   vpand %xmm2, %xmm14, %xmm2
0x52d529   353   vpunpckhdq %xmm3, %xmm4, %xmm10
0x52d52d   355   vpshufb %xmm2, %xmm13, %xmm1
0x52d532   355   vpsrlw $0xc, %xmm2, %xmm2
0x52d537   362   vpshufb %xmm0, %xmm4, %xmm4
0x52d53c   367   vpunpckldq %xmm0, %xmm10, %xmm0
0x52d540   355   vpshufb %xmm2, %xmm12, %xmm2
0x52d545   355   vpor %xmm2, %xmm1, %xmm2
0x52d549   355   vmovq %xmm2, %rdx
0x52d54e   363   vpshufb %xmm0, %xmm3, %xmm3
```

Highlighted 17 row(s):

Vector Code

Scalar Code [Including writes]

The Source of the problem

- 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];
```


DPDK
DATA PLANE DEVELOPMENT KIT

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*

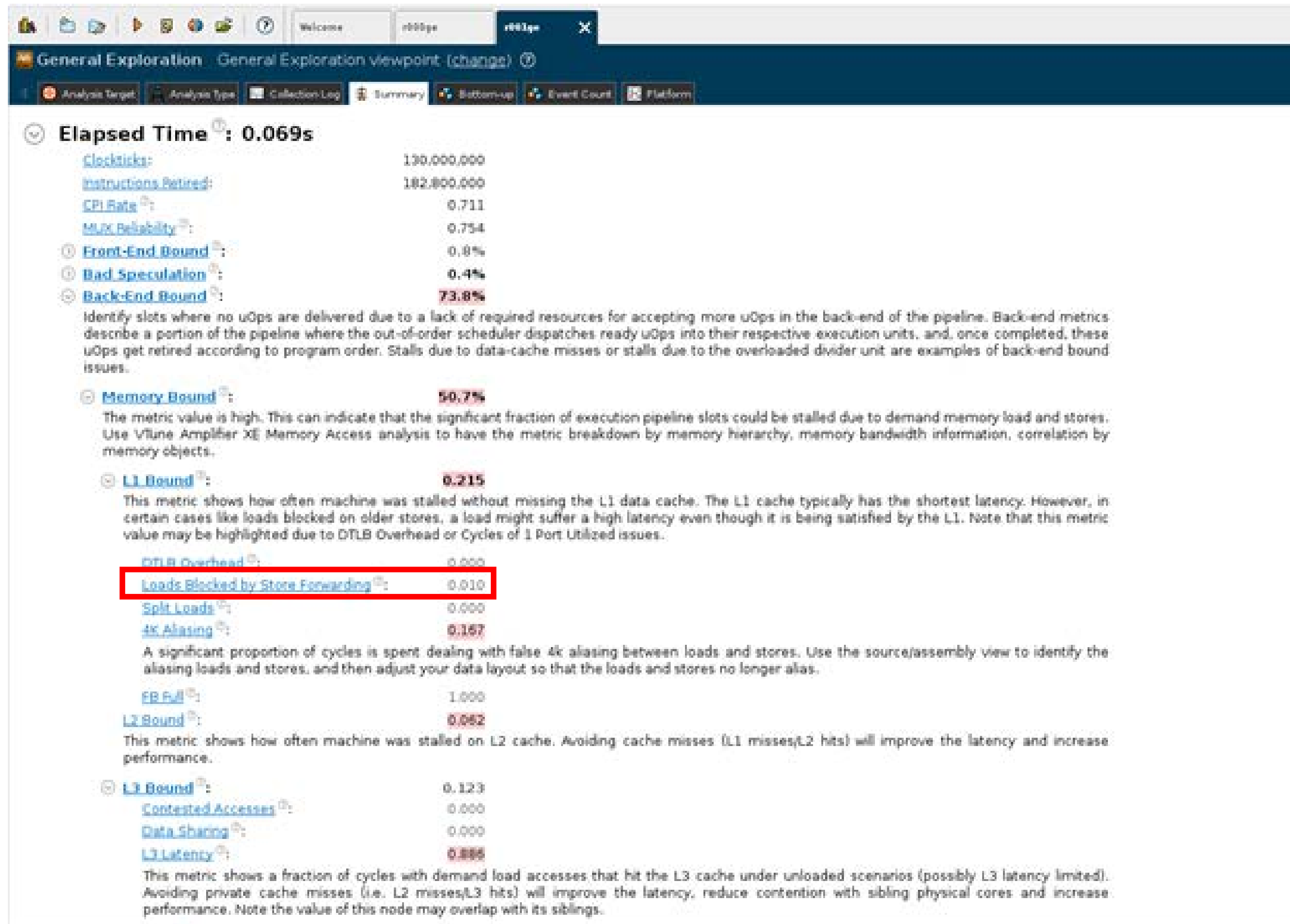
```
pktlen0 = _mm_srli_epi32(pktlen0, PKTLEN_SHIFT);
pktlen0 = _mm_and_si128(pktlen0, pktlen_msk);

pktlen0 = _mm_packs_epi32(pktlen0, pktlen0);

descs[3] = _mm_blend_epi16(descs[3], pktlen0, 0x80);
pktlen0 = _mm_slli_epi64(pktlen0, 16);
descs[2] = _mm_blend_epi16(descs[2], pktlen0, 0x80);
pktlen0 = _mm_slli_epi64(pktlen0, 16);
descs[1] = _mm_blend_epi16(descs[1], pktlen0, 0x80);
pktlen0 = _mm_slli_epi64(pktlen0, 16);
descs[0] = _mm_blend_epi16(descs[0], pktlen0, 0x80);
```

*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%.

DPDK
DATA PLANE DEVELOPMENT KIT