DP Benchmarking on Arm

NFV Data Plane Benchmarking @ ONS NA ‘18

Brian Brooks <brian.brooks@arm.com>
Tina Tsou <tina.tsou@arm.com>
Sirshak Das <sirshak.das@arm.com>
Use Case

VPP

- L2 cross connect
- IPv4 routing
- 64B @ 10Gbps
- Single flow & direction
- Single core

MUSDK

- L2 cross connect
- IPv4 routing
- 64B @ 10Gbps
- Single flow & direction
- Single core
Tracing Packets

How does the packet traverse the graph?
Runtime Clocks

ARMv8 Generic Timer
Not CPU clock cycles!

```
$ dmesg | grep MHz
[ 0.000000] Architected cp15 timer(s) running at 25.00MHz (phys).
[ 0.000001] sched_clock: 56 bits at 25MHz, resolution 40ns, wraps every 4398046511100ns
```
Batch Size

Does batch size affect performance?

I/O device *can* fill big frames
Identifying Hotspots

First access to packet data

Why is memory access the hotspot?

b0 = vlib_get_buffer (vm, bi0);
b1 = vlib_get_buffer (vm, bi1);

error0 = error1 = ETHERNET_ERROR_NONE;
e0 = vlib_buffer_get_current (b0);
type0 = c lib_net_to_host_u16 (e0->type);
e1 = vlib_buffer_get_current (b1);
type1 = c lib_net_to_host_u16 (e1->type);

prfm  pl dl1keep, [x0]  0.44
add   x0, x25,         0.03
      ldrh  w15, [x12,#12]  0.45
prfm  pl dl1keep, [x1]  59.48
str   x0, [x29,#592]    0.03
sub   w0, w28, #0x2     0.05
mov   w1, w15           10.05
str   w0, [x29,#576]    10.05
rev16 w1, w1
ldrh  w0, [x11,#12]    10.05
mov   v0.h[0], w1
rev16 w0, w0
mov   v1.h[0], w0
Dual-loop Explained

```c
void go_go_go(int *from, size_t num)
{
    while (num >= 4)
    {
        p0 = get_packet(from[0]);
        p1 = get_packet(from[1]);
        p2 = get_packet(from[2]);
        p3 = get_packet(from[3]);
        prefetch(p2->data);
        prefetch(p3->data);
        process(p0);
        process(p1);
        from += 2; num -= 2;
    }
    while (num > 0)
    {
        // ...
    }
}
```

Loop unrolled twice
Loop body interleaved
Prefetch 1 iteration ahead
I/O Memory Latency

- RX data serviced over DDR in I/O coherent system
- RX data serviced from cache via cache stashing
## Tuning Prefetches

<table>
<thead>
<tr>
<th>Stride</th>
<th>Dual Loop</th>
<th>Single Loop</th>
<th>Single Loop – Split PF</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Stride</td>
<td>Mpps</td>
<td>Clocks</td>
</tr>
<tr>
<td>+1</td>
<td>3.47</td>
<td>1.53e0</td>
<td></td>
</tr>
<tr>
<td>+2</td>
<td>3.74</td>
<td>1.13e0</td>
<td></td>
</tr>
<tr>
<td>+3</td>
<td>3.82</td>
<td>1.03e0</td>
<td></td>
</tr>
<tr>
<td>+4</td>
<td>3.78</td>
<td>1.04e0</td>
<td></td>
</tr>
<tr>
<td>+5</td>
<td>3.76</td>
<td>1.04e0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>S1</td>
<td>S2</td>
<td>Mpps</td>
</tr>
<tr>
<td>+1</td>
<td>+8</td>
<td>3.68</td>
<td>1.03e0</td>
</tr>
<tr>
<td>+2</td>
<td>+8</td>
<td>3.77</td>
<td>1.01e0</td>
</tr>
<tr>
<td>+3</td>
<td>+8</td>
<td>3.71</td>
<td>1.05e0</td>
</tr>
<tr>
<td>+2</td>
<td>+9</td>
<td>3.81</td>
<td>9.58e-1</td>
</tr>
<tr>
<td>+2</td>
<td>+10</td>
<td>3.81</td>
<td>9.58e-1</td>
</tr>
<tr>
<td>+2</td>
<td>+11</td>
<td>3.82</td>
<td>9.49e-1</td>
</tr>
<tr>
<td>+2</td>
<td>+12</td>
<td>3.84</td>
<td>9.24e-1</td>
</tr>
<tr>
<td>+2</td>
<td>+13</td>
<td>3.84</td>
<td>9.15e-1</td>
</tr>
<tr>
<td>+2</td>
<td>+14</td>
<td>3.82</td>
<td>9.37e-1</td>
</tr>
<tr>
<td>+3</td>
<td>+13</td>
<td>3.80</td>
<td>9.49e-1</td>
</tr>
</tbody>
</table>

Is load-to-use time the best we can “predict”?

- **61ns**
- **37ns**

Tune separately for L1 and L2 latencies.
Avoiding Bottlenecks

“The L1 memory system is non-blocking and supports hit-under-miss. For Normal memory, up to six 64-byte cache line requests can be outstanding at a time. While those requests are waiting for memory, loads to different cache lines can hit the cache and return their data.”


Prefetching combined with loop unrolling is demanding!
Types of Data Accesses

Device descriptor

```
b->total_length_not_including_f =
b->flags = VLIB_BUFFER_TOTAL_LE
mov w8, pp2_ppio_inq_desc_get_pkt_len();
  ldrh w0, [x21,#6]
mrvl_pp2_set_buf_data_len_flags()
mrvl_pp2_set_buf_data_len_flags()
```

<table>
<thead>
<tr>
<th>Frame Size</th>
<th>Vector (4B)</th>
<th>Descriptor (32B)</th>
<th>Buffer (128B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>.25KB</td>
<td>2KB</td>
<td>8KB</td>
</tr>
<tr>
<td>128</td>
<td>.5KB</td>
<td>4KB</td>
<td>16KB</td>
</tr>
<tr>
<td>256</td>
<td>1KB</td>
<td>8KB</td>
<td>32KB</td>
</tr>
<tr>
<td>512</td>
<td>2KB</td>
<td>16KB</td>
<td>64KB</td>
</tr>
</tbody>
</table>

vlib_buffer_t

```
d += 2;
buffers += 2;
n_desc -= 2;
strh w1, [x29,#360]
  if (n_trace > 0)
    mrvl_pp2_input_trac
```

vlib_physmem_region_t

```
add  x1, x0, 
11.98  ldrb  w0, [x0,#22]
add  x0, x7, x0, lsl
14.23  ldrb  w0, [x0,#24]
umaddl x0, w0, w9, x10
27.81  ldr   x14, [x0,#8]
```
Initial Remarks

Observations

• Most hotspots are memory accesses
• Software-defined data placement consumes processing cycles
• Unintentionally ordering memory accesses can slow the system down
• Compiler may fuse loops which alters memory access pattern from original program order

Further Directions

• Leverage PMU data
• Compiler and C library versions
• Multicore scaling
• Platforms
  • Cavium, Huawei, Qualcomm, ...
Preliminary Results

64B packet – single flow – single core

Mpps

<table>
<thead>
<tr>
<th></th>
<th>Basline</th>
<th>New</th>
</tr>
</thead>
<tbody>
<tr>
<td>l2xc</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ip4-routing</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Baseline: 1.31x
New: 1.39x

% line rate

<table>
<thead>
<tr>
<th></th>
<th>l2xc</th>
<th>ip4-routing22</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td></td>
<td></td>
</tr>
<tr>
<td>128</td>
<td></td>
<td></td>
</tr>
<tr>
<td>256</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The path to on Arm

<table>
<thead>
<tr>
<th>Workload Scale</th>
<th>Performance Analysis</th>
<th>Software</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSIT</td>
<td>Hotspot &amp; Bottleneck Identification</td>
<td>Upstream</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Libraries</td>
</tr>
<tr>
<td></td>
<td></td>
<td>OS</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Toolchain</td>
</tr>
<tr>
<td>FD.io Lab</td>
<td>Tuning &amp; Optimization</td>
<td>Hardware</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Processors</td>
</tr>
<tr>
<td></td>
<td></td>
<td>I/O</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Accelerators</td>
</tr>
</tbody>
</table>
The Path to **FD.io** on Arm

- **Workload Scale**
  - Continue integration of Arm-based platforms into FD.io lab
  - Adopt and run CSIT on a diverse range of machines and topologies

- **Performance Analysis**
  - Distill critical runtime components affecting performance
  - Identify solutions to hotspots and/or bottlenecks

- **Upstream**
  - Integrate solutions back into open source
Thank You!
Danke!
Merci!
谢谢!
ありがとう!
Gracias!
Kiitos!
감사합니다
धन्यवाद
```shell
gcc (Ubuntu/Linaro 5.4.0-6ubuntu1~16.04.9) 5.4.0 20160609

ldd (Ubuntu GLIBC 2.23-0ubuntu10) 2.23

$ cat /proc/cmdline
console=ttyS0,115200 root=/dev/sda1 rw

$ cat /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages
16

$ lscpu
Architecture:          aarch64
Byte Order:            Little Endian
CPU(s):                4
On-line CPU(s) list:   0-3
Thread(s) per core:   1
Core(s) per socket:    2
Socket(s):             2
CPU max MHz:           2000.0000
CPU min MHz:           100.0000
Hypervisor vendor:     (null)
Virtualization type:   full
```
“Debugging is the act of asking questions and answering them, not guessing what the answer is.

You want to form questions, not hypotheses. Answers to questions constrain hypotheses.

We repeat this process. Specific questions.. Specific answers.. More specific questions.. More specific answers.. And then.. that ‘hypothetical leap’ is often not a leap at all. It’s a step across a puddle.

That is how we debug. We debug by having the cycle of questions and answers.

We are not magicians. We are the wizard of Oz sweating behind a curtain frenetically turning a crank trying to figure out the problem.”

Bryan Cantrill

*Debugging Under Fire: Keep your Head when Systems have Lost their Mind*