EiovaR
Efficient IOMMU Intra-Operating System Protection

Moshe Malka, Nadav Amit, and Dan Tsafrir
Technion - Israel Institute of Technology
IOMMU = I/O memory management unit
IOVA = I/O virtual address
Direct memory access is dangerous
IOMMU provides memory protection

**Intra-OS protection**

- Physical memory
- Device A
- Device B
- IOMMU

**Inter-OS protection**

- Physical memory
- Device
- Guest A
- Guest B
- Virtual machines
How Intra-OS protection works (strict)

1) allocates an IO Virtual Address

2) Updates the page table

Before the DMA
How Intra-OS protection works (strict)

access the memory using virtual address

main memory

page table

IOVA allocator

IOTLB
How Intra-OS protection works (strict)

1) Removes the translation from page table

After the DMA

2) Free the IOVA

3) Invalidate the IOTLB entry
strict vs. defer
Protection tradeoff

1) removes the translation from page table
2) free the IOVA
3) Invalidate the IOTLB entry

After the DMA
what’s the performance cost?

NIC (mlx4) throughput without IOMMU

$\frac{1500 \cdot 8 \cdot \text{frequency}}{\text{cycles}}$
what’s the performance cost?

NIC throughput with IOMMU (mlx4)

1500·8·frequency/cycles

throughput [Gbps]

cycles per packet (C)

none

measured w/ iommu

model

strict
what’s the performance cost?

NIC throughput with IOMMU (mlx4)

\[
\text{throughput} = \frac{1500 \cdot \text{frequency}}{\text{cycles}}
\]
Why is IOMMU so costly?

• Common wisdom: “HW is to blame!”
• Correlation between the # of cycles it takes to process a packet and the throughput
• Our claim: SW is guiltier!
  • OSes drive IOMMUs with suboptimal software
  • Could be much improved
  • All OSes we examined (not just Linux)
Overhead breakdown - same bench as before

- Overhead breakdown
  - 9.8x
  - 4.9x
  - Cycles per packet
  - iova
  - iotlb inv
  - page table
  - other

- Strict
  - Completely safe

- Defer
  - Tradeoff

- No iommu
Why is IOVA (de)allocation so costly?

IOVA allocation

Cached node

Virtual address space

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>90</td>
<td>91</td>
<td>92</td>
<td>93</td>
<td>94</td>
</tr>
</tbody>
</table>
Why is IOVA (de)allocation so costly?

IOVA allocation after ..98 is reallocated

Cached node

Virtual address space

search starts here

search ends here
EiovaR: efficient IOVA allocator

On IOVA allocation

1) Pop an IOVA
2) If not exist: allocate from the regular tree

Regular IOVA data structure (tree)

On IOVA free

Push the IOVA to the corresponding list

Lists of free IOVA ranges

<table>
<thead>
<tr>
<th>2↑</th>
<th>2↑</th>
<th>2↑</th>
<th>...</th>
<th>...</th>
<th>...</th>
<th>2↑</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>31</td>
</tr>
</tbody>
</table>
Results
How many cycles we reduced?
How many cycles we reduced?
Performance improvement

NIC throughput with IOMMU (mlx4)

throughput [Gbps]

cycles per packet (C)

model

measured w/ iommu

1500·8·frequency/cycles

none
defer
strict
Performance improvement

NIC throughput with IOMMU (mlx4)

throughput [Gbps]

cycles per packet (C)

none

ASPLOS 2015

defer+

defer

strict+

strict

model

measured w/ iommu

20

15

10

5

0

2k

4k

6k

8k

10k

12k

14k

16k

18k
## Micro- and macro-benchmarks (mlx)

<table>
<thead>
<tr>
<th>protection</th>
<th>benchmark</th>
<th>baseline</th>
<th>EiovaR</th>
<th>diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>strict</td>
<td>Netperf Stream</td>
<td>1</td>
<td>2.37</td>
<td>+137%</td>
</tr>
<tr>
<td></td>
<td>Netperf RR</td>
<td>1</td>
<td>1.27</td>
<td>+27%</td>
</tr>
<tr>
<td></td>
<td>Apache 1MB</td>
<td>1</td>
<td>3.65</td>
<td>+265%</td>
</tr>
<tr>
<td></td>
<td>Apache 1KB</td>
<td>1</td>
<td>2.35</td>
<td>+135%</td>
</tr>
<tr>
<td></td>
<td>Memcached</td>
<td>1</td>
<td>4.58</td>
<td>+358%</td>
</tr>
<tr>
<td>defer</td>
<td>Netperf Stream</td>
<td>1</td>
<td>1.71</td>
<td>+71%</td>
</tr>
<tr>
<td></td>
<td>Netperf RR</td>
<td>1</td>
<td>1.07</td>
<td>+7%</td>
</tr>
<tr>
<td></td>
<td>Apache 1MB</td>
<td>1</td>
<td>1.21</td>
<td>+21%</td>
</tr>
<tr>
<td></td>
<td>Apache 1KB</td>
<td>1</td>
<td>1.11</td>
<td>+11%</td>
</tr>
<tr>
<td></td>
<td>Memcached</td>
<td>1</td>
<td>1.25</td>
<td>+25%</td>
</tr>
</tbody>
</table>
Generalizing

• So far, all results are related to Linux
• We found, however, that other OSes similarly drive the IOMMU with suboptimal SW
• In the paper, we show that FreeBSD is worse than Linux
  • (We even describe a patch that we already managed to push to FreeBSD that improves throughput by 76 %)
• We also found that Solaris and Darwin have similar deficiencies (not in the paper)
Conclusions

• Common wisdom is “IOMMU is slow”
  • Not true!
  • Suboptimal SW makes it much slower than it should be

• We find that characteristics of the workload experienced by the IOMMU allow for even our basic & minimal freelist optimization to deliver high performance (which BTW consists of about 250-lines patch)

• Although our analysis focuses on Linux, other OSes suffer from issues
Thank You!
Q & A