A scalable, common IOMMU pooled allocation library for multiple architectures

Sowmini Varadhan (sowmini.varadhan@oracle.com)

Agenda

- What is IOMMU?
- Benefits/drawbacks of IOMMU
- Typical design of IOMMU allocators
  - Extracting common constructs: a generic IOMMU pooled allocator
  - Some optimizations, performance results
- Some case-studies of sun4v/sun4u usage
- Ongoing/future Work

What is IOMMU?

- IOMMU = I/O Memory Management Unit
- Connects a DMA capable I/O bus to the main memory
  - IOMMU maps bus-address space to the physical address space.
  - IOMMU is the bus-address mapper, just as MMU is the virtual address mapper
  - IOMMU and MMU have many of the same benefits and costs.

Pictorially..

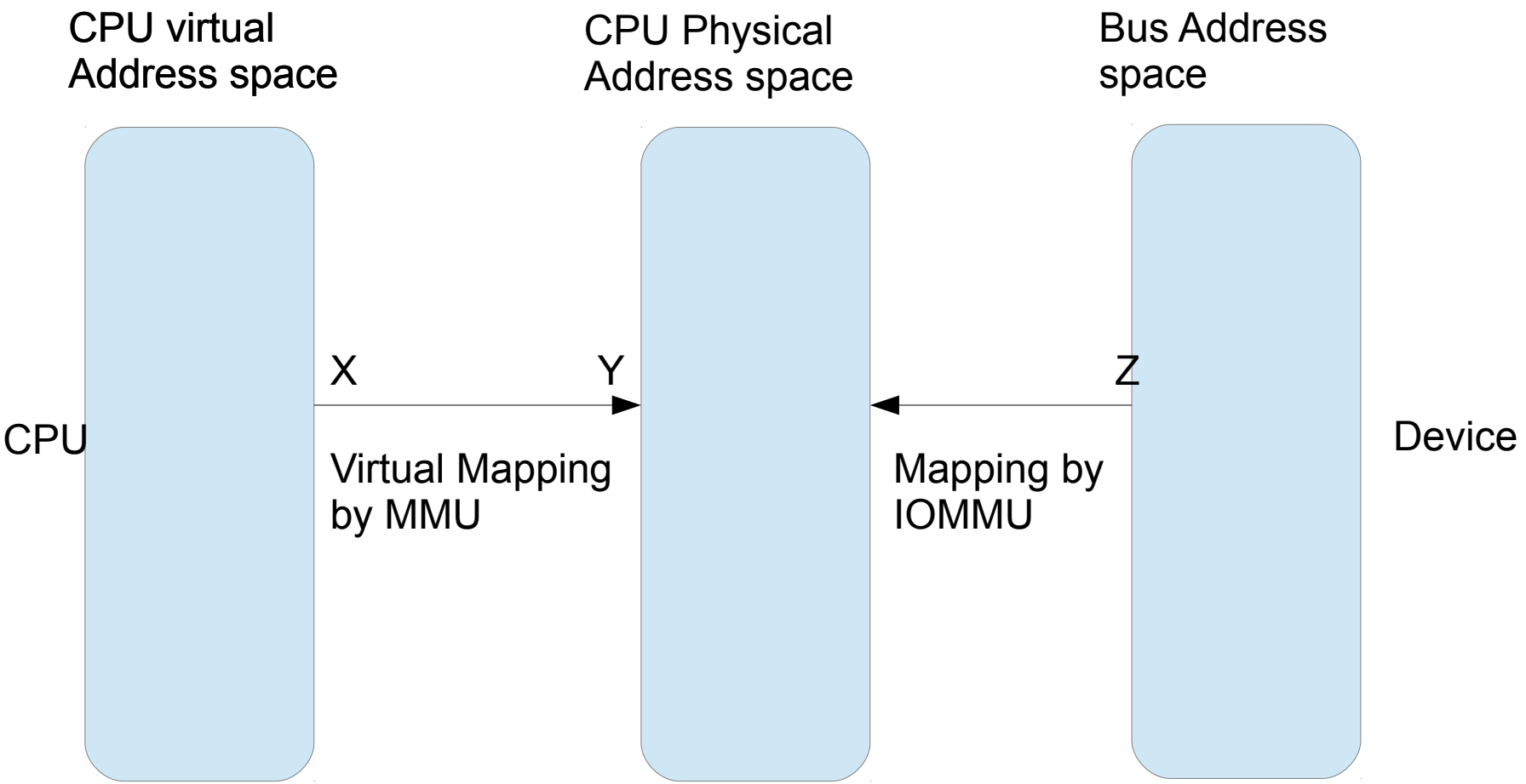


Diagram adapted from Documentation/DMA-API-HOWTO.txt

Why have IOMMU?

- Bus address length != Physical address size
  - e.g., bus address width may be 24 bits, physaddr may be 32 bit. Remapping allows the device to efficiently reach every part of physical memory. Without IOMMU, inefficient schemes like bounce buffers would be needed.
- Can efficiently map a contiguous bus address space to non-contiguous main-memory.

Additional IOMMU Benefits

- Protection: a device cannot read or write memory that has not been explicitly mapped for it.
- Virtualization: guest OS can use h/w that is not virtualization aware: IOMMU handles re-mapping needed for these devices to access the vaddrs remapped as needed by the VM subsystem.

IOMMU drawbacks..

- Extra mapping overhead
- Consumption of physmem for page tables

IOMMU and DMA

- Virtual address “X” is generated by kmalloc()
- VM system maps “X” to a physaddr “Y”
  - But a driver cannot set up DMA to the bus address for “Y” without the help of IOMMU
- IOMMU translates “Z” to “Y”
- To do DMA, driver can call `dma_map_single()` with “X” to set up the needed IOMMU for “Z”

How does this work?

Driver calls a mapping indirection from its `dma_map_ops` e.g., `->map_page`. The driver passes in the virtual address it wishes to DMA to, and the number of pages (`'n'`) desired.

- IOMMU allocator now needs to set up the bus-addr ↔ physaddr mappings as needed for DMA.

DMA mapping and IOMMU allocation

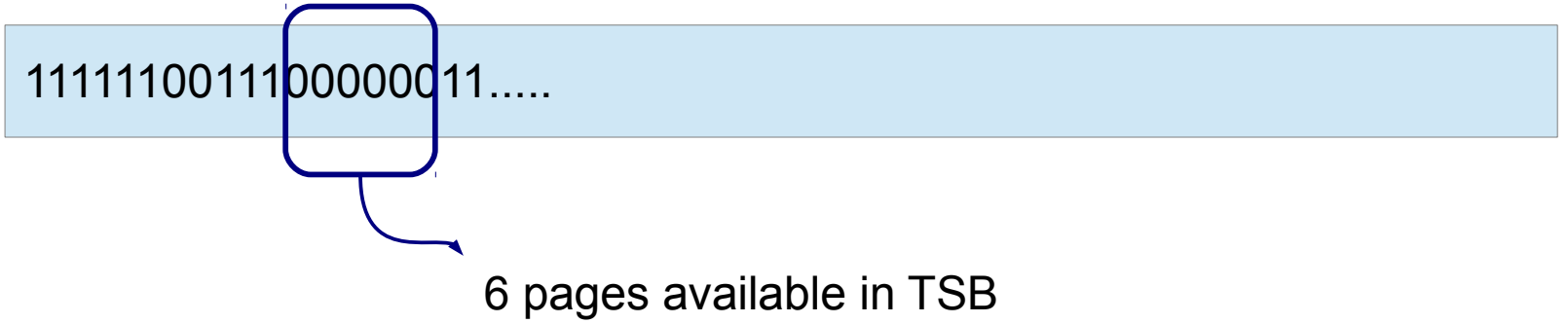
- Given virtual address (`vaddr`) for the page passed in, and an allocation request for `n` pages..
- Call the IOMMU arena allocator to find an available block of `n` entries in the vdma/TSB for the device (TSB = Translation Storage Buffer)
- For those entries returned (indices into TSB) commit the physaddr of the `vaddr` mapping in the TSB

IOMMU arena allocation

- The IOMMU allocator needs to have a data-structure representing the current state of the device TSB
- Typically represented as a bitmap, where each bit represents a page of the bus address space. A bit value of “1” indicates that the page is in use.

IOMMU Arena Allocator

- Finding a block of `n` entries <=> finding a contiguous set of `n` zero-bits.



IOMMU area allocation

- Simplistic algorithm:

```
mutex_lock(&iommu->lock)
/* find a zero area of n bits */
/* mark area as "taken" */
mutex_unlock(&iommu->lock)
/* update TSB with physaddrs */
```

Optimized IOMMU Arena Allocator

- We don't need the left-most zero-area, any contiguous set will suffice
- Can parallelize the arena allocation by dividing bitmap into pools

- Only need to lock each pool, i.e., per-pool lock.
- 4 threads can concurrently call arena allocator

Results: ethernet perf before optimization

Before the IOMMU lock optimization,

- Iperf with 8-10 threads, with TSO disabled, can only manage 1-2Gbps on a 10 Gbps point-to-point ethernet connection on a sparc T5-2 (64 CPUs, 8 cores/socket, 8 threads/core)
- Lockstat shows about 35M contentions, 40M acquisitions for the `iommu->lock`
- Lock contention points: `dma_4v_[un]map_sg, dma_4v_[un]map_page`

Ethernet perf results after lock optimization

After lock fragmentation (16 pools from 256K TSB entries)

- Can achieve 8.5-9.5 Gbps without any additional optimizations, both with and without TSO support.
- Lockstat now only shows some slight contention on the iommu lock: approx 250 contentions versus 43M acquisitions

Results: RDS over Infiniband

- "rds-stress" request/response test: IOPS (transactions per second) for 8K sized RDMA on a T7-2 (320 CPUs, 2 sockets, 12 cores/socket, 13 threads/core).
  - "single pool" represents unoptimized allocator

	Single pool	16 pools
1 conn	80K	90K
4 conn	80K	160K

Additional enhancements

- Breaking up the bitmap into pools may result in fragmented pools that cannot handle requests for a very large number of pages.
- Large pool support (from PPC): upper ¼ of the bitmap is set up as a “large pool” for allocations of more than `iommu_large_alloc` pages (default = 15)

Additional enhancements

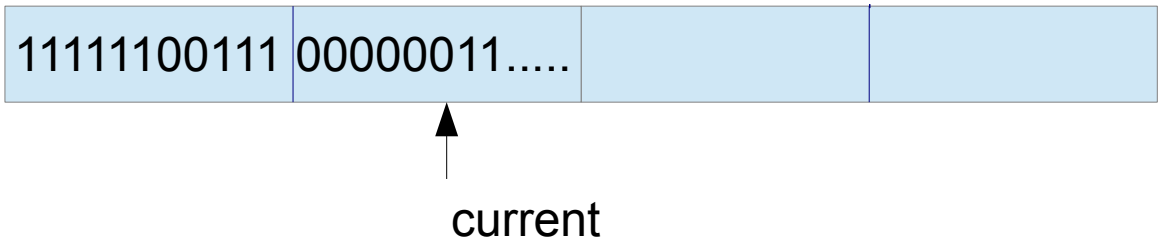
- Breaking up the bitmap into pools may result in fragmented pools that cannot handle requests for a very large number of pages.
- Large pool support (from PPC): upper ¼ of the bitmap is set up as a “large pool” for allocations of more than `iommu_large_alloc` pages (default = 15)

Pool initialization parameters: large pool support

- In practice, large pool allocations are rare, and dependant on the type of device
  - Make large-pool reservation an optional parameter during IOMMU initialization via `iommu_tbl_pool_init()`
- `iommu_large_alloc` cut-off should also be a pool parameter?

Optimizing TSB flushes

- When the arena allocator returns, it tracks the current location where the search terminated.



- Next allocator invocation would start at current, and search toward the right end of the pool, wrapping around if necessary
  - i.e., circular buffer search.

lazy_flush()

- Arena allocator hands out entries in a strict lowest → highest order within the pool. So if an entry to the left of "current" has not been re-used, there is no need to flush the TSB entry either.
- Leverage this where possible (e.g., older sun4u h/w), by only flushing when we go back to the beginning of the pool in the arena allocator.
- Caller of pool init must supply the `lazy_flush()` where this is possible

sun4v initializing the IOMMU allocator

- Core data-structure used by library is struct `iommu_map_table` which has the bitmap "`map`"
- `iommu_map_table` initialization from `pci_sun4v_iommu_init:`

```
iommu->tbl.map = kzalloc(sz, ..);
```

- Call iommu_tbl_pool_init() to initialize the rest of `iommu_map_table`, with `npools == 16`, no large pool, null (*lazy_flush)()

sun4v IOMMU alloc

- First call iommu_tbl_range_alloc() to find the indices of the TSB that have been set aside for this allocation
- Commit the physaddr ↔ bus-addr mappings to the TSB by invoking `pci_sun4v_iommu_map()` (HV call)

sun4v: releasing an IOMMU allocation

- First reset the TSB state by calling
  `pci_sun4v_iommu_demap()` (HV call)
- Then reset iommu bitmap data-structure by calling
  iommu_tbl_range_free()
  - MUST hold the lock on the specific pool from which allocation was made

sun4u customizations

- No intermediate HV, so the `lazy_flush` is non-null.
- Currently uses only 1 pool for 1M TSB entries- need to experiment with using multiple pools
  - Caveat: typical sun4u is 2 socket, 2 CPU.

Current/ongoing Work

- Currently: **all** the sparc code uses the generic iommu allocator: includes LDC (for Ldoms), sun4u, sun4v
- Convert PPC and x86_64 code to use the common arena allocator
- Try using multiple pools on older hardware (sun4u) and check for perf improvements

Acknowledgements

- David Miller, for the original sparc code, and encouragement to create a generic library,
- Ben Herrenschmidt and PPC team for all the reviews and feedback that helped make this better,
- All the folks on sparclinux that helped test this on legacy sun4u platforms,
- Linuxcon, Oracle for their support.