An Implementation of Fast memset() Using Hardware Accelerators

Runtime and Operating Systems for Supercomputers Workshop

June 12 2018

Kishore Pusukuri

NetApp Inc

Rob Gardner

Oracle Inc

Jared Smolens

Arm Inc

Multicore Systems & Big-memory Applications

- Multicore systems with huge caches and many cores are ubiquitous
  - ➢ e.g., SPARC T7-1: 32 cores (256 vCPUs), 64MB L3 cache, and 512GB RAM
  - ➢ Maximizes performance of emerging *big-memory* workloads (databases, graph analytics, key-value stores, and HPC workloads)
- Big-memory applications
  - ➢ Require many virtual-to-physical address translations in page tables and TLBs
  - ➢ e.g., consumes ~51% of execution cycles just on TLB misses [ISCA'13]

[ISCA'13] A. Basu et al., Efficient Virtual Memory for Big Memory Servers. In Proceedings of ISCA, 2013.

Huge Pages

- Modern hardware and OS introduced support for huge pages
  - e.g., Linux (kernel 2.6.39) on T7 supports 8MB, 2GB, and 16GB (default 8KB)
  - Improves performance and system utilization by reducing TLB miss rate
- Creating a huge page is an expensive operation
  - Need to zero the page through kernel memset()
  - e.g., while creating a 2GB page takes *322 msec*, zeroing it takes *320 msec*
  - Zeroed during kernel initialization (booting) and whenever application requests
  - Big-memory applications need 100s of huge pages → zeroing operation impacts kernel initialization times (application startup times/serviceability)

This Work

On a SPARC T7-1 running Oracle Linux (2.6.39)

- Exploits T7 and its co-processors (DAX) to speed up the creation of huge pages by up to 11x → improves database startup time up to 6x
- Presents an enhanced memset() that uses T7 co-processors → improves JVM Garbage Collector latencies by 4x

Outline

1. Background
2. DAX memset
3. Hybrid DAX memset
4. Case Studies

Kernel memset()

During kernel initialization

80 Million Times

- 99.9% calls are for zeroing

Zeroing Huge Pages

- **`clear_huge_page()`** takes 320 msecs to zero a 2GB huge page

```
void clear_huge_page(struct page *page,
                    unsigned long addr,
                    unsigned int pages_per_huge_page
                    )
{
        ...
        for (i = 0; i < pages_per_huge_page; i++) {
                cond_resched();

                // calls kernel memset()
                clear_user_highpage(page + i, addr + i * PAGE_SIZE);
        }
}
```

Improving clear_huge_page()

- Kernel Threads
- Work Queues (a pool of worker threads)

Technique	8MB page	2GB page
Kernel Threads	2.0x	4.9x
Work Queues	4.0x	5.0x

- Multithreaded kernel memset()
  - Complexity of using multiple kernel threads? (Software Engineering)
  - Spawning multiple kernel threads on a loaded system? (Performance)

SPARC T7 & DAX (1)

- T7 processor has 8 co-processor units (DAX)
- Data is directly read from and written to the memory space
- Multiple tasks (or commands) can be simultaneously submitted and hypervisor controls queuing and scheduling
- DAX calls are asynchronous (mwait instruction can be used to monitor the status → facilitates heterogeneous computing)

SPARC T7 & DAX (2)

- How to request DAX?
  - Fill in a ccb (coprocessor control block) with the operation to be done, pointers to various memory regions, sizes, etc.
  - Call hypervisor API: hypervisor distributes ccb commands (tasks or DAX requests) across DAX units

- Only works on contiguous memory -- page boundaries cannot be crossed by a single command (task)
- Max. ccb blocks per ccb_command (or DAX request) are 15
- Max. concurrent ccb_commands are 64

Outline

1 SPARC M7 DAX

2 DAX memset

3 Hybrid DAX memset

4 Case Studies

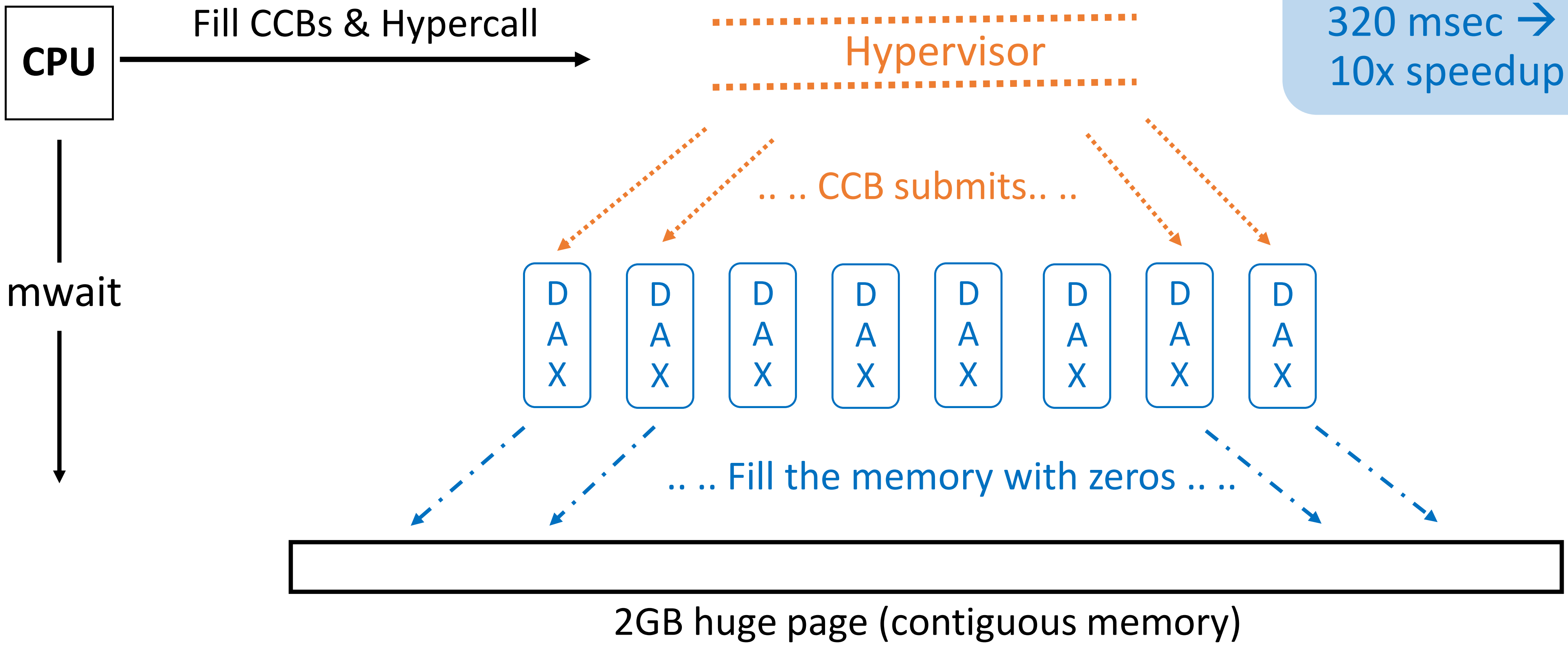
DAX memset()

- Assumes contiguous memory (effective for huge pages)
- As memset(address, fill_value, size), where [address..address+size] must reside on one page

```
/* fill coprocessor control block (ccb) */
ccb_fill_t ccb;
….
ccb.tl.hdr.opcode  = CCB_MSG_OPCODE_FILL;
ccb.ctl.hdr.at_dst = CCB_AT_VA;
ccb.imm_op         = fill_value;
ccb.ctl.size       = size;
...

/* ccb_submit to DAX */
hypercall(completion_area, &ccb, …);
…..
mwait()
....
```

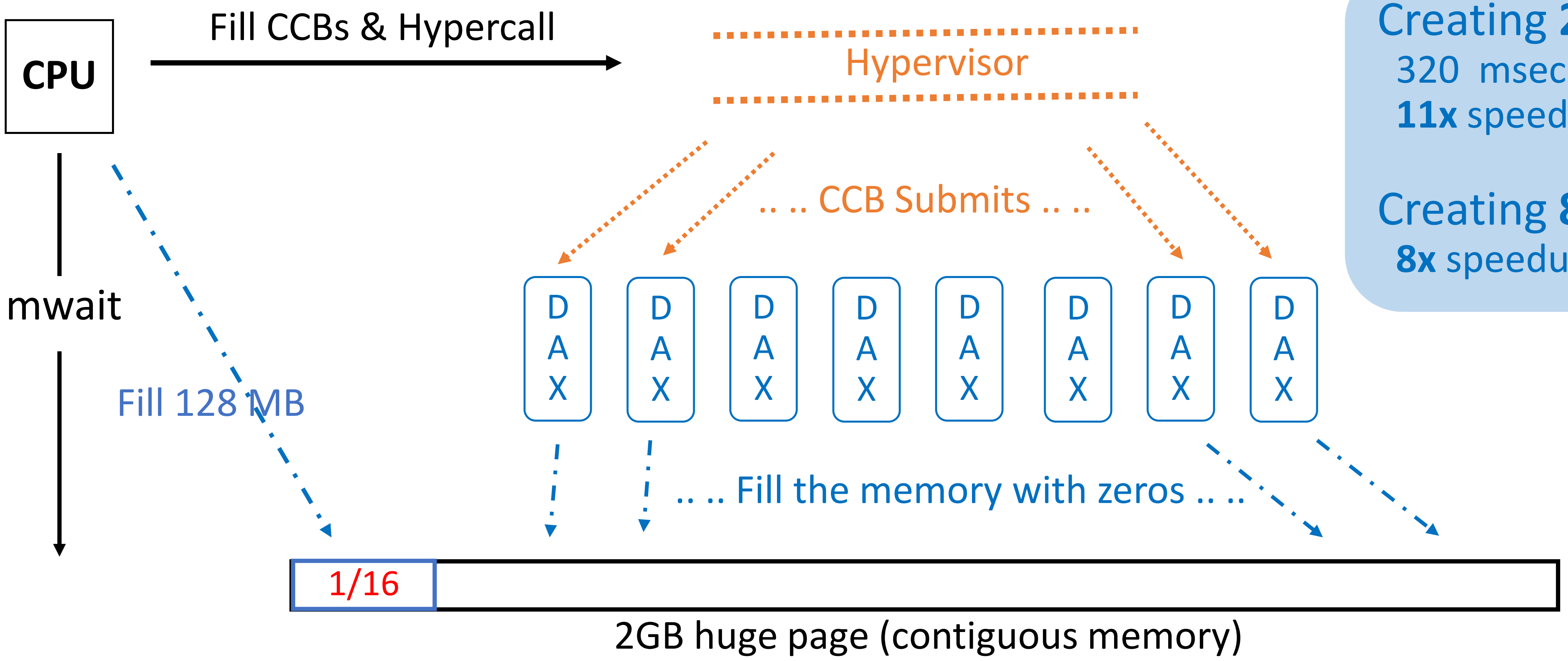
DAX memset() (cont...)



Creating 2GB:
320 msec → 31 msec,
10x speedup

DAX memset() (cont...)

Heterogeneous computing



Creating **2GB**:
320 msec→ 29 msec,
11x speedup

Creating **8MB**:
8x speedup

DAX memset() vs Multithreaded Techniques

Technique	8MB page	2GB page
Kernel Threads	2.0x	4.9x
Work Queues	4.0x	5.0x
DAX	8.0x	11.0x

Outline

1. SPARC M7 DAX
2. DAX memset()
3. Hybrid DAX memset()
4. Case Studies

Hybrid DAX memset (1)

- Kernel memset() functionality: should also work on non-contiguous memory (assume no huge pages)

- Identify the discontinuities in physical memory corresponding to the entire virtual buffer – need to do a page table lookup for each page
- Derive a scatter-gather list: a list of contiguous memory chunks (starting address of the memory region and its length)
  - e.g., { [0x100, 200]; [0x400, 100]; [0x1000, 200]; [0x6000, 400]; [0x2000, 300]}
- Feed each item of the list as a CCB to the DAX
  - Max. ccbs per ccb_command (or task) are 15 and Max. condurrent ccb_commands are 64

Hybrid DAX memset (2)

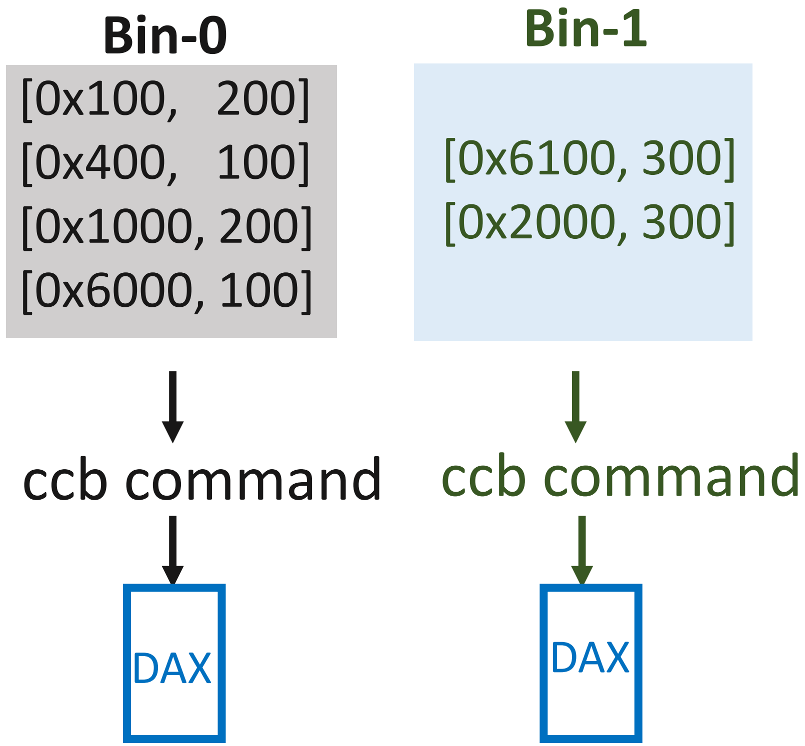
- Effectively distribute the scatter-gather list across all the 8 DAX units -- balancing load across DAX units

Scatter-gather list:
[Address, length]
{
R0: [0x100, 200]
R1: [0x400, 100]
R2: [0x1000, 200]
R3: [0x6000, 400]
R4: [0x2000, 300]
}

Bin-packing Algorithm:
number of bins = 2
max capacity = 1200/2 = 600

Bin-0 ← **R0** + **R1** + **R2** + **R3** (100)
Bin-1 ← **R3** (300) + **R4** (300)

6.5x speedup for 32 MB size
- Default memset: 6504 usec
- Hybrid DAX memset : 1008 usec



Optimizing Hybrid DAX memset (1)

- The overhead (va_to_pa) of generating scatter-gather list is 402 usec (for 32 MB)

- Derive scatter-gather list in two stages
  - Process half of the memory and build the scatter-gather list (then derive bins)
  - Feed the bins (ccb_commands) to DAX units
  - Use CPU to process second half of the memory while DAX is processing the ccb_commands of the first half of memory

Speedup: 6.5x → 7.9x

- Default memset : 6504 usec
- DAX MEMSET : 1008 usec
- DAX MEMSET (opt-1) : 821 usec

Optimizing Hybrid DAX memset (2)

- Fill 1/8th of memory region using CPU (while DAX is processing 7/8th)

Speedup: 6.5x → 8.7x

- Default memset()	:	6504 usec
- Hybrid DAX memset()	:	1008 usec
- Hybrid DAX memset() (opt-1)	:	821 usec
- Hybrid DAX memset() (opt-1 + opt-2)	:	746 usec

- Achieves 9x speedup in zeroing 32MB memory

Case Studies

- Database SGA Preparation Times (Startup Times)
  - improves up to 5x (256GB SGA, with 8MB pages)

- Java JVM GC Latency
  - improves by up to 4.0x (provided ioctl() interface for applications)

Limitations

- Only effective for sizes > 256KB as the DAX setup takes ~15 usec
- Contention for DAX devices when the load is heavy?

Conclusions

- Demonstrates the potential benefits of utilizing T7 coprocessors
  - Speeds up the creation of huge pages by 11.0x
  - Java JVM GC latencies are improved by up to 4.0x

This Work

On a SPARC T7-1 running Oracle Linux (2.6.39)

- Exploits T7 and its co-processors (DAX) to speed up the creation of huge pages by up to 11x → improves database startup time up to 6x
- Presents an enhanced memset() that uses T7 co-processors → improves JVM Garbage Collector latencies by 4x