System Software for Persistent Memory

Subramanya R Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran and Jeff Jackson

72131715 Neo Kim
phoenixise@gmail.com

Contents

Contents

Abstract

- PMFS
  - File system for accessing PM with low performance overhead
  - Legacy applications are supported without changes.
  - Light-weight POSIX file system
  - Byte-addressability enables direct PM access with memory-mapped I/O.
  - Guarantee consistency with a simple hardware primitive
  - Memory protection from stray writes
  - Performance is evaluated using a hardware emulator.

Contents

Introduction

- ## Persistent Memory
  - Large capacity, byte-addressable, storage class memory
  - Even though the performance gap is brought down, PM is still accessed as block device which causes unnecessary overheads.

- ## PMFS
  - Implements a file system for accessing PM without block layer
  - Complies with POSIX interface to support for legacy applications
  - Light-weight file system and optimized memory-mapped I/O because of no need to copy data between DRAM and storage

Introduction

- **Challenges**
  - Ordering and durability
  - Protection from stray writes
  - Validation and correctness testing of consistency

- **Contributions**
  - A simple new hardware primitive
  - Design and implementation
    - A light-weight POSIX file system
    - Fine-grained logging for consistency
    - Direct mapping of PM to application
    - Low overhead scheme for protecting PM from stray writes

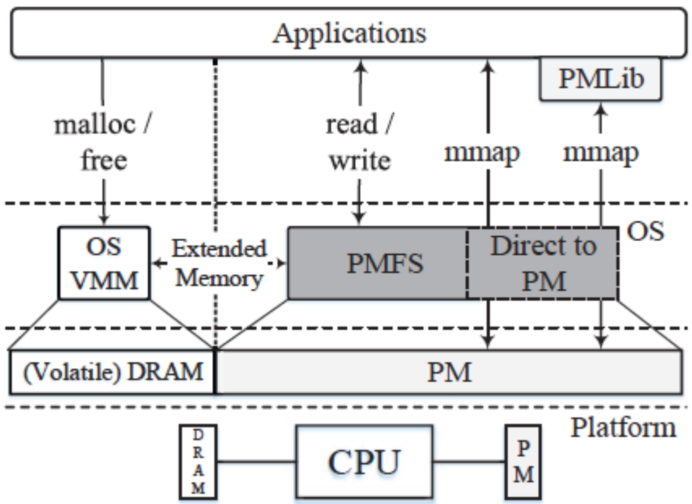
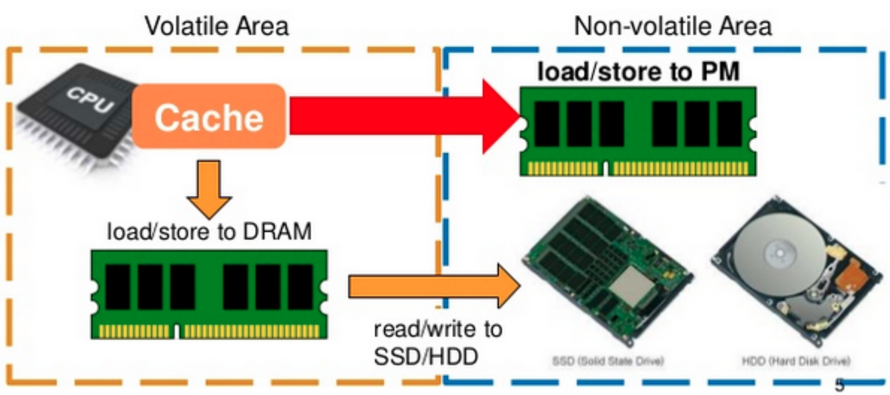


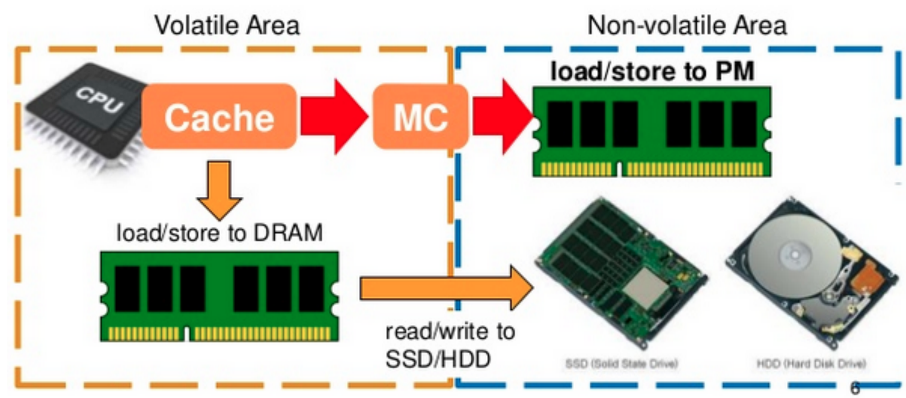
Figure 1: PM System Architecture

Contents

System Architecture

- ## Consistency - ordering and durability
    - Approaches
        - PM as write-through
        - PM writes bypassing the CPU cache (non-temporal)
        - Epoch base ordering
    - Using PM as write-back cacheable and flushing works well.
    - To reach the durability point, propose pm_wbarrier.





Contents

Design and Implementation

- ## Goals
  - Optimization for byte-addressable storage
    - To avoid copies to DRAM during file I/O
  - Enable efficient access to PM by applications
  - Protect PM from stray writes

- ## Memory-mapped I/O
  - PMFS Layout
  - Memory-mapped I/O
    - Chooses largest page table

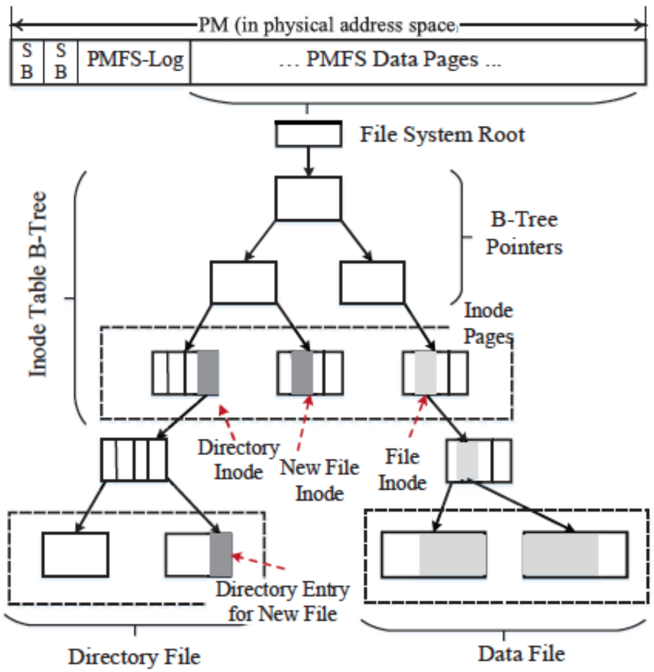


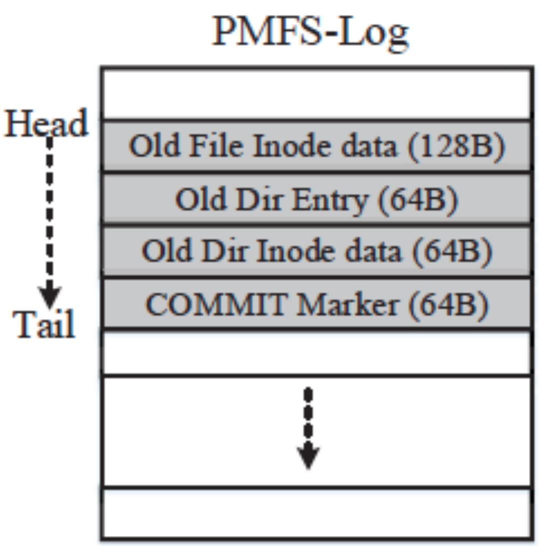
Figure 3: PMFS data layout

Design and Implementation

- ## Consistency
  - Possible techniques
    - CoW, journaling, log-structured updates
  - CoW, log-structured file systems performs at block or segment size granularity
    - Metadata updates are small so large granularity causes large write amplification
  - Journaling can log the metadata updates at finer granularity
  - Found that 64-byte (cacheline) granularity incurs the least overhead for metadata updates.
  - Use CoW for file data updates.

Design and Implementation

- **Journaling**
  - Redo journaling
    - The new data is logged and made durable. Once transaction committed, the new data is written to the file system.
    - Pros - Needs only 2 pm_wbarriers
    - Cons - Reads during transaction need to search the log entries
  - Undo journaling
    - The old data to be overwritten is logged and made durable. And the new data is written to the file system during transaction.
    - Pros - Simple and fine-granularity is possible
    - Cons - One pm_wbarrier for every log entry



(a)

Design and Implementation

- ## Atomic in-place updates
  - Update the metadata directly without using logging.
  - PMFS leverages processor features for 8, 16, 64-byte atomic updates
    - 8-byte - Use to update inode access time on file read
    - 16-byte - Use to update inode size and modification time when appending
    - 64-byte - Use to modify a number of inode fields

Design and Implementation

- ## Write Protection
  - Corruption can be occurred due to bugs in unrelated software (stray write)
  - How write protection works
    - Row name refers to the address space
    - Column name refers to the privilege level

	User	Kernel
User	Process Isolation	SMAP
Kernel	Privilege Levels	Write windows

Table 2: Overview of PM Write Protection

  - Protection “kernel from user” - by Privilege levels
  - Protection “user from user” - by Paging
  - Protection “user from kernel” - by SMAP(Supervisor Mode Access Prevention)
    - Supervisor-mode (ring 0 or kernel) accesses to the user address space are not allowed

Design and Implementation

- **Write Protection (..continued)**
  - Protection “kernel from kernel” - Write windows
    - Map the entire PM as read-only during mount
    - Upgrades it to writeable only for the sections of code that write to PM

```
P: Read-only PM page in kernel virtual address
write(P): Write to page P in ring 0 (kernel)
GP: General protection fault
```

```
// CR0.WP in x86

if (ring0 && CR0.WP == 0)
   write(P) is allowed;
else
   write(P) causes GP;
```

```
// Using CR0.WP in PMFS
disable_write_protection() {
   CR0.WP = 0;
   disable_interrupts();
}
enable_write_protection() {
   enable_interrupts();
   CR0.WP = 1;
}
```

```
// Writes to PM in PMFS
disable_write_protection();
write(P);
enable_write_protection();
```

Figure 5: Write Protection in PMFS

Design and Implementation

Testing and Validation

- Maintaining consistency is challenging.
- PM software needs to track dirty cachelines, to flush them explicitly before issuing pm wbarrier.
- The same concern applies to any PM software.
- To address this issue built Yat, which is framework to help validate PM
- Yat operates in two phases
  - Records a trace of all the writes, clflush, ordering, pm_wbarrier
  - Replays the collected traces and test all subsets and orderings

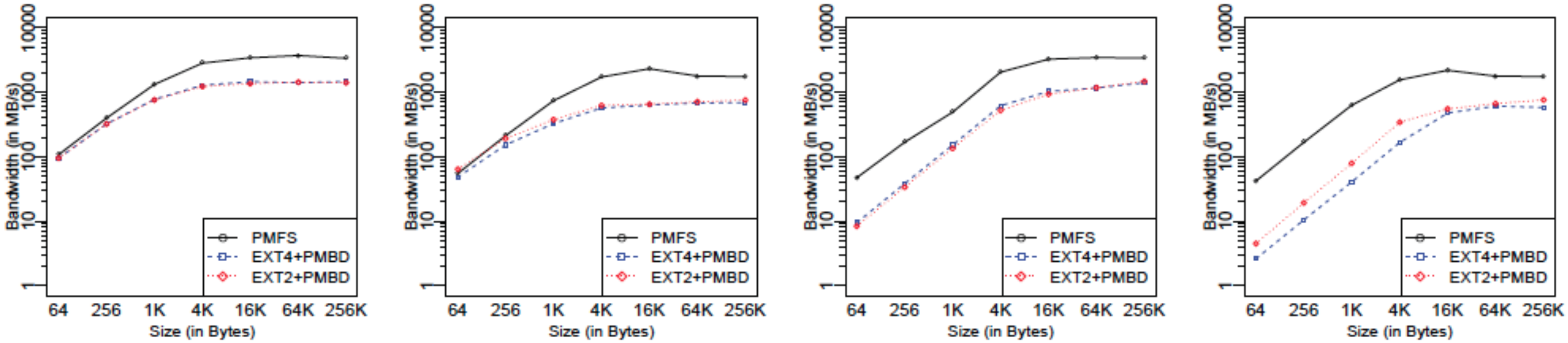
Contents

Evaluation

- ## Experimental Setup
  - PM Emulator
    - System-level evaluation of PM software is challenging due to lack of real hardware.
    - Built PMEP(PM Emulation Platform)
    - Partitions the available DRAM memory into emulated PM and regular volatile memory
    - Emulate PM latency
    - Emulate PM bandwidth
  - PMBD
    - Use PMBD for a fair comparison with block devices

Evaluation - File-based Access

- File-based I/O



a) Sequential read b) Sequential write c) Random read d) Random write

Figure 6: Evaluation of File I/O performance; X-axis is the size of the operation; Y-axis is the bandwidth in MB/s.

  - For all the tests, with improvements ranging from 1.1× (for 64B sequential reads) to 16× (for 64B random writes).
  - The drop in writes for sizes larger than 16KB is due to the use of non-temporal instructions. These instructions bypass the cache but still incur the cache-coherency overhead

Evaluation - File-based Access

● Consistency

- Atomic in-place updates to avoid logging
  - write system call using 16-byte atomic updates - up by 1.8x
  - delete an inode using 64-byte atomic updates - 18% faster
- Logging overhead - benefits of fine-grained logging
  - For PMFS and ext4, measured the amount of metadata logged
  - For BPFS, measured the amount of metadata copied (using CoW).

	PMFS	BPFS (vs PMFS)	ext4 (vs PMFS)
touch	512	12288 (24x)	24576 (48x)
mkdir	320	12288 (38x)	32768 (102x)
mv	384	16384 (32x)	24576 (64x)

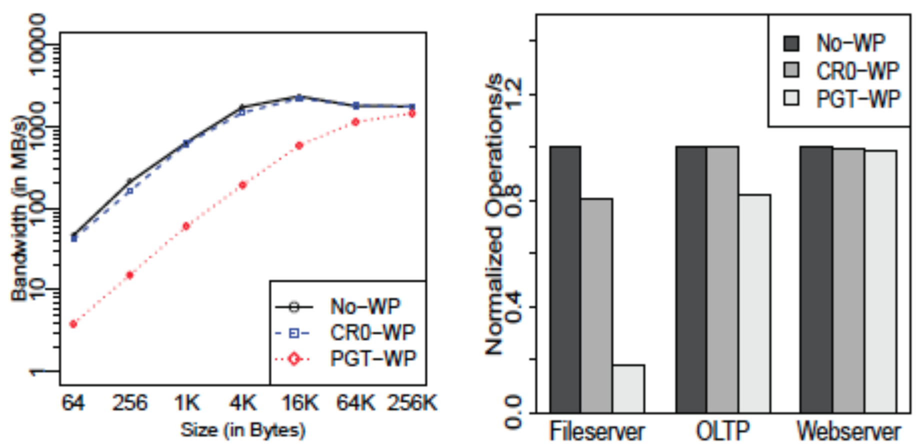
Table 3: Metadata CoW or Logging overhead (in bytes)

Evaluation

- ## Memory-mapped I/O
  - Neo4j Graph Database
    - By default, Neo4j accesses files by mmap
    - The graph has 10M nodes and 100M edges.
    - Compared to ext2, ext4, 1.1x (Insert) to 2.4x (Query)

Evaluation

- Write Protection
  - Compared to No-WP, PGT-WP(page table permission)
  - File server - workload with writes from several threads (23% slower)
  - OLTP - a single log thread
  - Webserver - less write intensive workload



a) *fio* file write b) Filebench

Figure 11: Evaluation of PMFS write protection overhead


I REGRET NOTHING!

References

- Instruction of System Software for Persistent Memory, http://www.slideshare.net/makotoshimazu/readingcircle-20141218
- Wear Leveling, http://en.wikipedia.org/wiki/Wear_leveling
- Write Amplification, http://en.wikipedia.org/wiki/Write_amplification
- Write-back cache - http://www.webopedia.com/TERM/W/write_back_cache.html
- Cache line - http://stackoverflow.com/questions/3928995/how-do-cache-lines-work
- Out-of-order Execution, http://en.wikipedia.org/wiki/Out-of-order_execution