DFS: A Filesystem for Virtualized Flash Disks

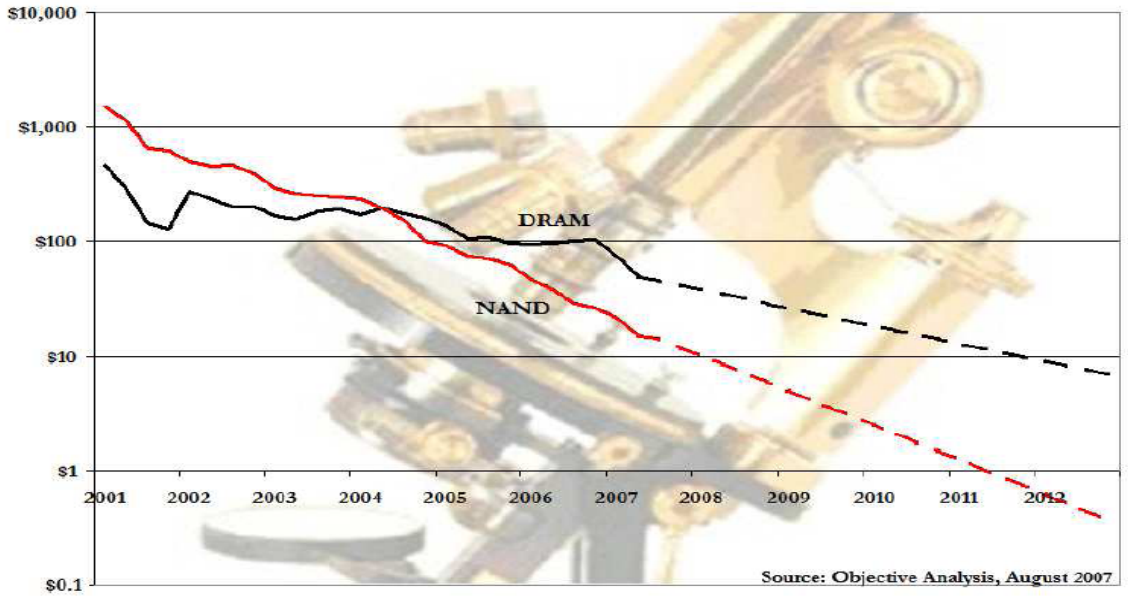
William K. Josephson and Lars A. Bongo,

David Flynn, Fusion-io; Kai Li, Princeton University

FAST 2010

Presenter: Wucherl Yoo

Motivation - Flash



- Flash costs less than DRAM and getting cheaper
- Many file systems designed for disk – FFS, XFS, FAT
  - Flash Translation Layer of SSD - indirection from logical block to flash page, wear leveling, copying for performance
- Flash file systems designed for embedded apps – JFFS, YAFFS
  - Small size, manage raw level of flash

Background – Flash Memory

- **Non-volatile Solid State Memory**
  - Update requires erase then re-write
  - Limited # of erase/write cycles - 1,000 to 10,000 per cell for MLC, 100,000 per cell for SLC
  - NOR – random access, read speed
  - NAND
    - Sequential access (µs), cheaper cost, slow random write
    - Data is organized into “pages” for transfer (512B-4KB)
    - Pages are grouped into “erase blocks” (16KB-16MB+)
      - 2ms for 256KB
  - SLC – single level cell, single bit per cell
  - MLC – multi-level cell, multiple bits per cell
- **FusionIO IODrive**
  - SLC NAND flash array connected via PCI-Express

Challenges - NAND Flash

- **Requirements**
  - Read/Write multiple pages
  - Erase entire erasure block
  - Update copied to empty erasure block
  - Wear-leveling
  - Error correction mechanism
  - HW Parallelism and SW support for performance

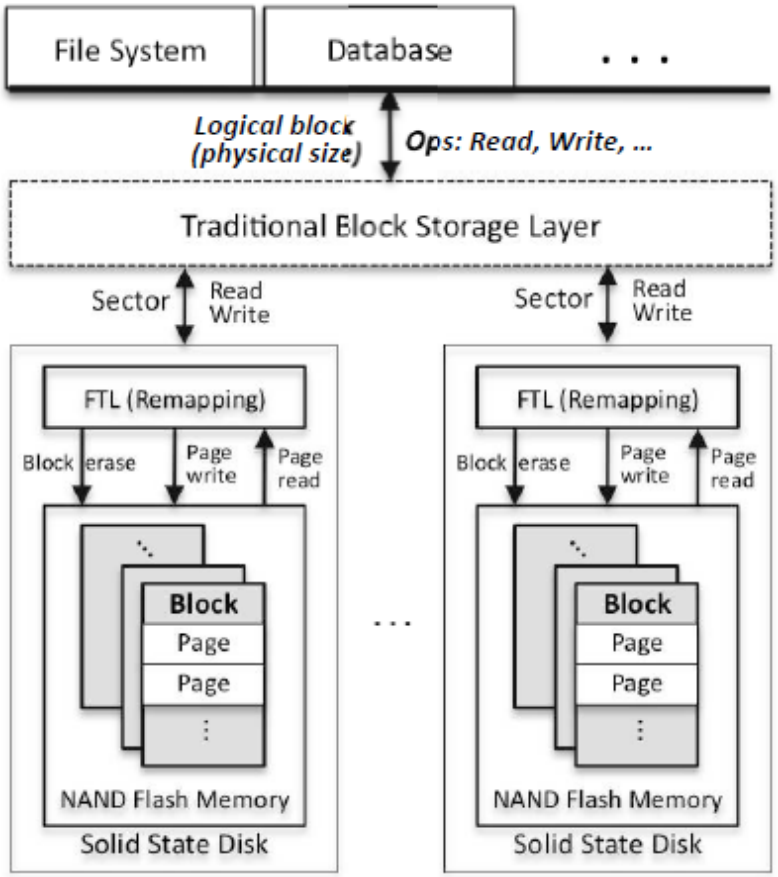
Design

- **Virtualized Flash Storage**
  - Large virtual block-address space – mapping to flash page
    - Backward-compatibility for block interface
    - 64bit virtual address, 512 byte block
  - File block allocations/reclamations
    - Wear leveling/bulk erasure
  - Atomic flash block updates for crash recovery
    - Write ahead log for every write for a single flash block, group commit
    - Dependencies among metadata and data?
  - Garbage collector – discard a block or block range

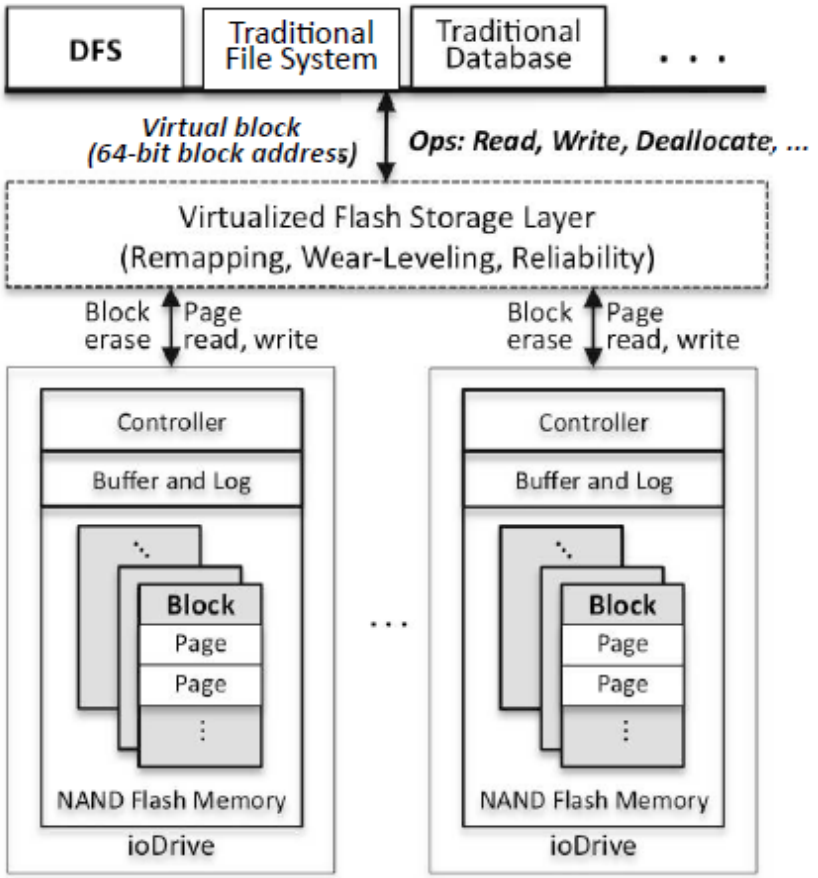
Design (2)

- **Direct File System (DFS)**
  - Backward compatibility with traditional block interface
  - Directory management with FFS metadata – requires additional logging of directory update
  - Combine multiple small I/O requests to adjacent regions into a single larger I/O

Flash Storage Abastraction

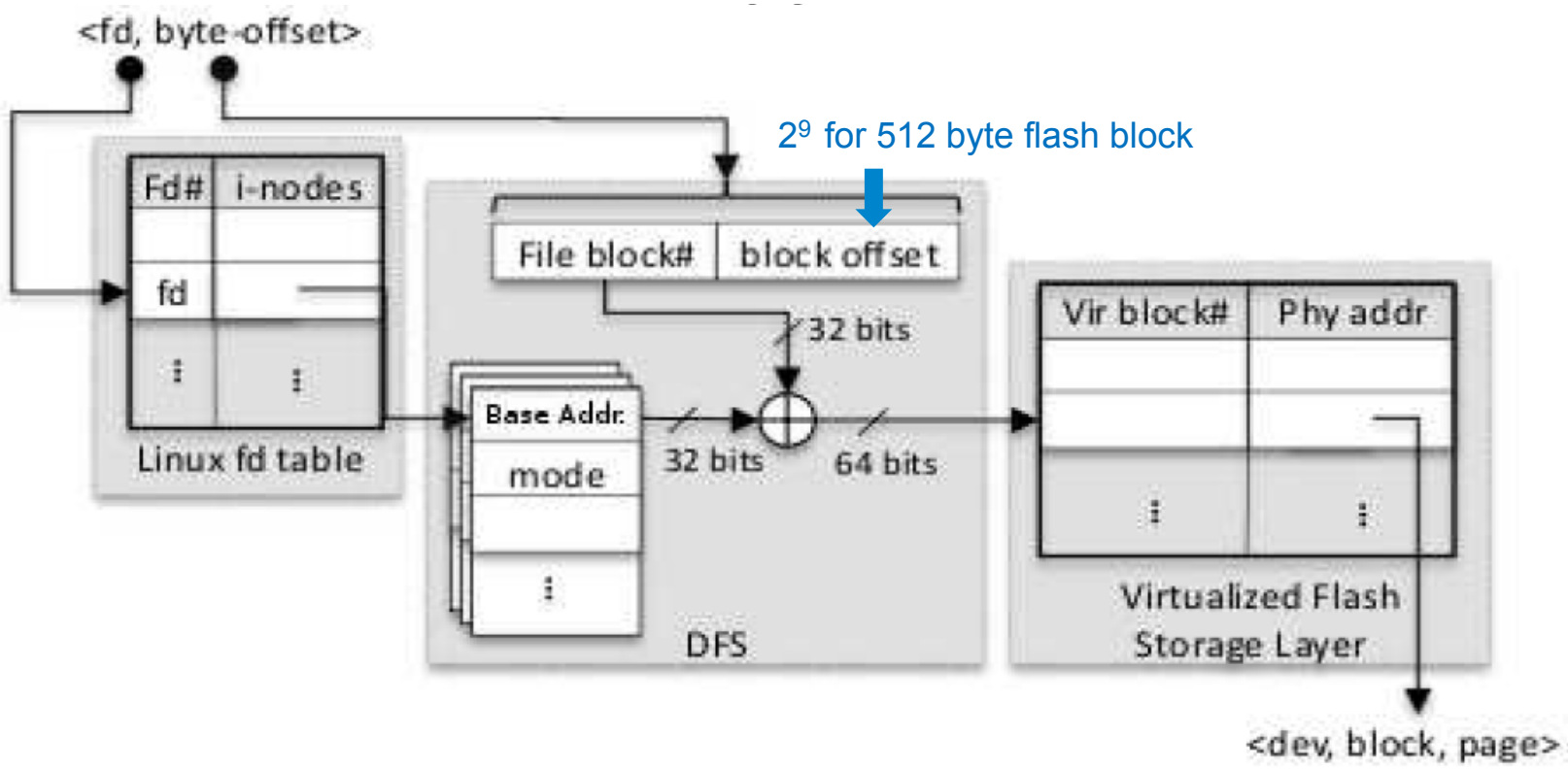


(a) Traditional layers of abstractions



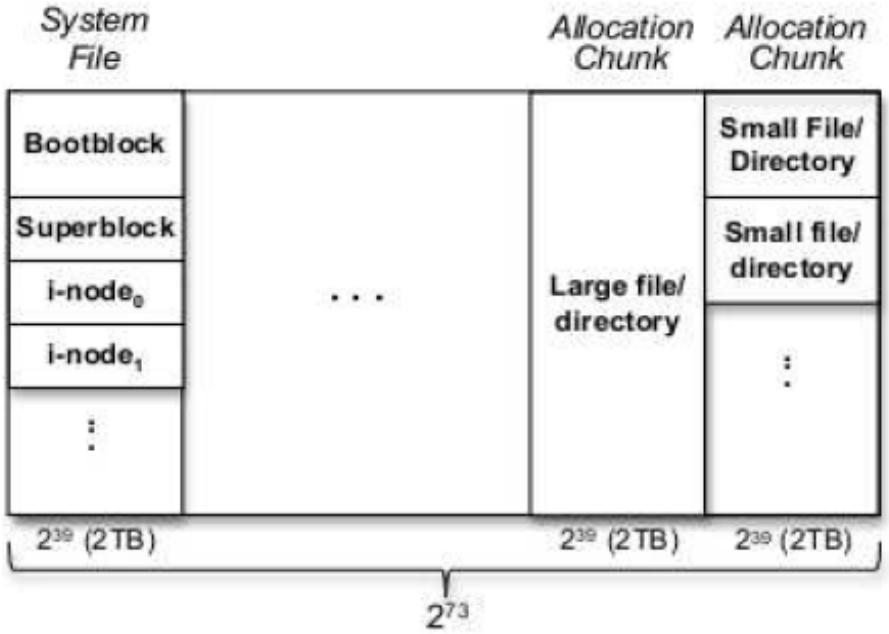
(b) Our layers of abstractions

DFS – Logical Block Address Mappi



- I-node – stored in 512 byte block, contains base virtual address
  - Increased I-node size can reduce dependency (only use 72 bytes)
- Virtual Address (Base addr, logical block #, block offset)
  -> Physical Address (dev, block, page)

DFS – File System Layout

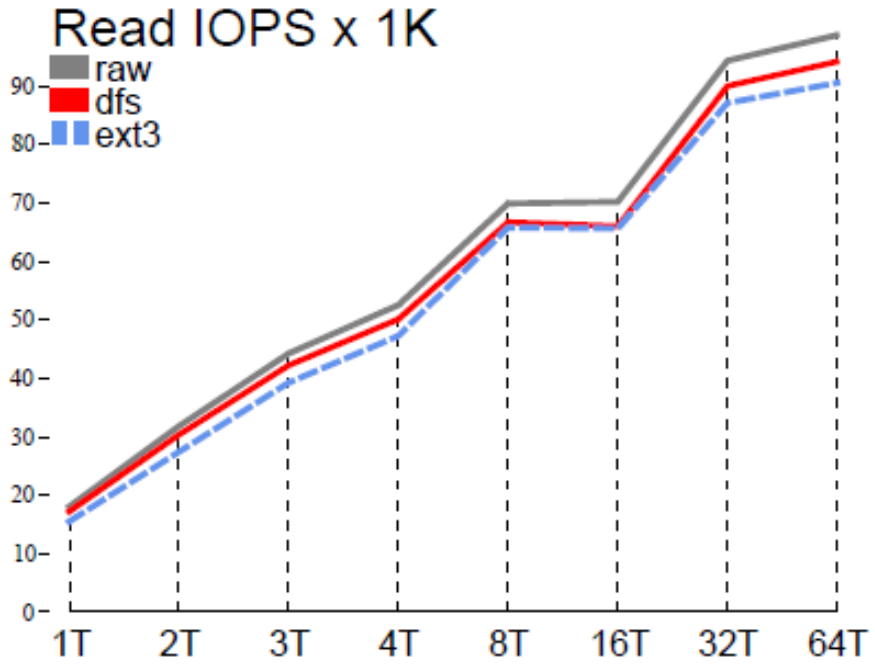


- System File Chunk – bootblock, superblock, i-nodes
- Allocation Chunk - 32-bit block-addressed
  - Small or Large file
  - Size chosen at initialization (max size of small file also)
- Metadata Update – write ahead log for recovery
  - Unclear how to handle dependencies among blocks

Experimental Environments

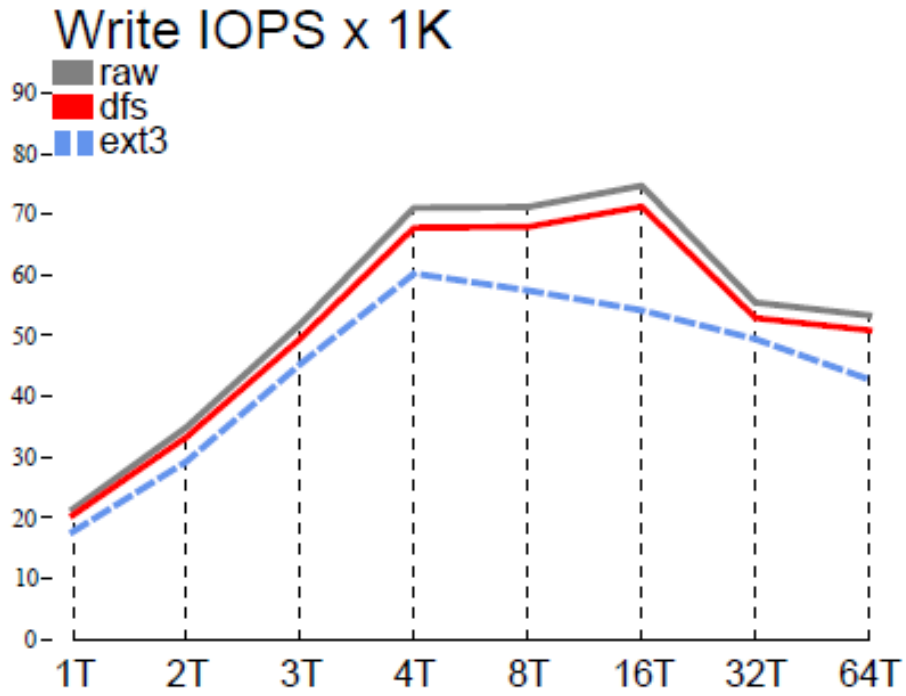
- **Environments**
  - Intel Quad Core 2.4 GHz, 4GB DRAM
  - FusionIO ioDrive with 160GB SLC NAND flash
    - Read latency - 50µs
    - Theoritical maximum throughput for single reader – 20,000 IOPS
    - Device driver exports block device interface

Microbenchmark –Random Read



- 4KB I/O transactions (IOZone)
- Multiple threads utilizes parallelism from flash
- DFS performance close to raw level

Microbenchmark – Random Write



- 4KB I/O
- Peak at 16T – overhead on the write path

Microbenchmark - CPU

Threads	Read	Random Read	Write	Random Write
1	8.1	2.8	9.4	13.8
2	1.3	1.6	12.8	11.5
3	0.4	5.8	10.4	15.3
4	-1.3	-6.8	-15.5	-17.1
8	0.3	-1.0	-3.9	-1.2
16	1.0	1.7	2.0	6.7
32	4.1	8.5	4.8	4.4

- Little Improvement for Write with Small # of Threads
- Garbage Collector Overhead – 4 cores

Application Benchmark

Applications	Description	I/O Patterns
Quicksort	A quicksort on a large dataset	Mem-mapped I/O
N-Gram	A hash table index for n-grams collected on the web	Direct, random read
KNNImpute	Missing-value estimation for bioinformatics microarray data	Mem-mapped I/O
VM-Update	Simultaneous update of an OS on several virtual machines	Sequential read & write
TPC-H	Standard benchmark for Decision Support	Mostly sequential read

Application Benchmark (2)

Application	Read IOPS x 1000		Write IOPS x 1000	
	Ext3	DFS (Change)	Ext3	DFS (Change)
Quick Sort	1989	1558 (0.78)	49576	1914 (0.04)
N-Gram (Zipf)	156	157 (1.01)	N/A	N/A
KNNImpute	2387	1916 (0.80)	2686	179 (0.07)
VM Update	244	193 (0.79)	3712	1144 (0.31)
TPC-H	6375	3760 (0.59)	52310	3626 (0.07)

Threads	Wall Time in Sec.		Ctx Switch x 1K	
	Ext3	DFS	Ext3	DFS
1	10.82	10.48	156.66	156.65
4	4.25	3.40	308.08	160.60
8	4.58	2.46	291.91	167.36
16	4.65	2.45	295.02	168.57
32	4.72	1.91	299.73	172.34

N-Gram(Zipf)

Application Benchmark (3)

Application	Wall Time		
	Ext3	DFS	Speedup
Quick Sort	1268	822	1.54
N-Gram (Zipf)	4718	1912	2.47
KNNImpute	303	248	1.22
VM Update	685	640	1.07
TPC-H	5059	4154	1.22

- **DFS Speedup**
  - Lower file lock contention – per block i-node, write-ahead log instead of journal
  - Aggregation of I/O request – smaller number with larger size

Discussion Points

- **What improves performance of DFS?**
  - Aggregation, parallelization, simple implementation
  - More CPU, garbage collection, increased I-node size
  - Reduced consistency from journal, lack of atomic multi-block update
- **Is virtual/physical mapping useful abstraction?**
  - Large logical space can be easily shared by distributed systems
  - Fixed first chunk, translation overhead
- **Is flash fit to cloud?**
  - Cost/Power efficient, Scalable to multi-threads
  - Wear lifetime