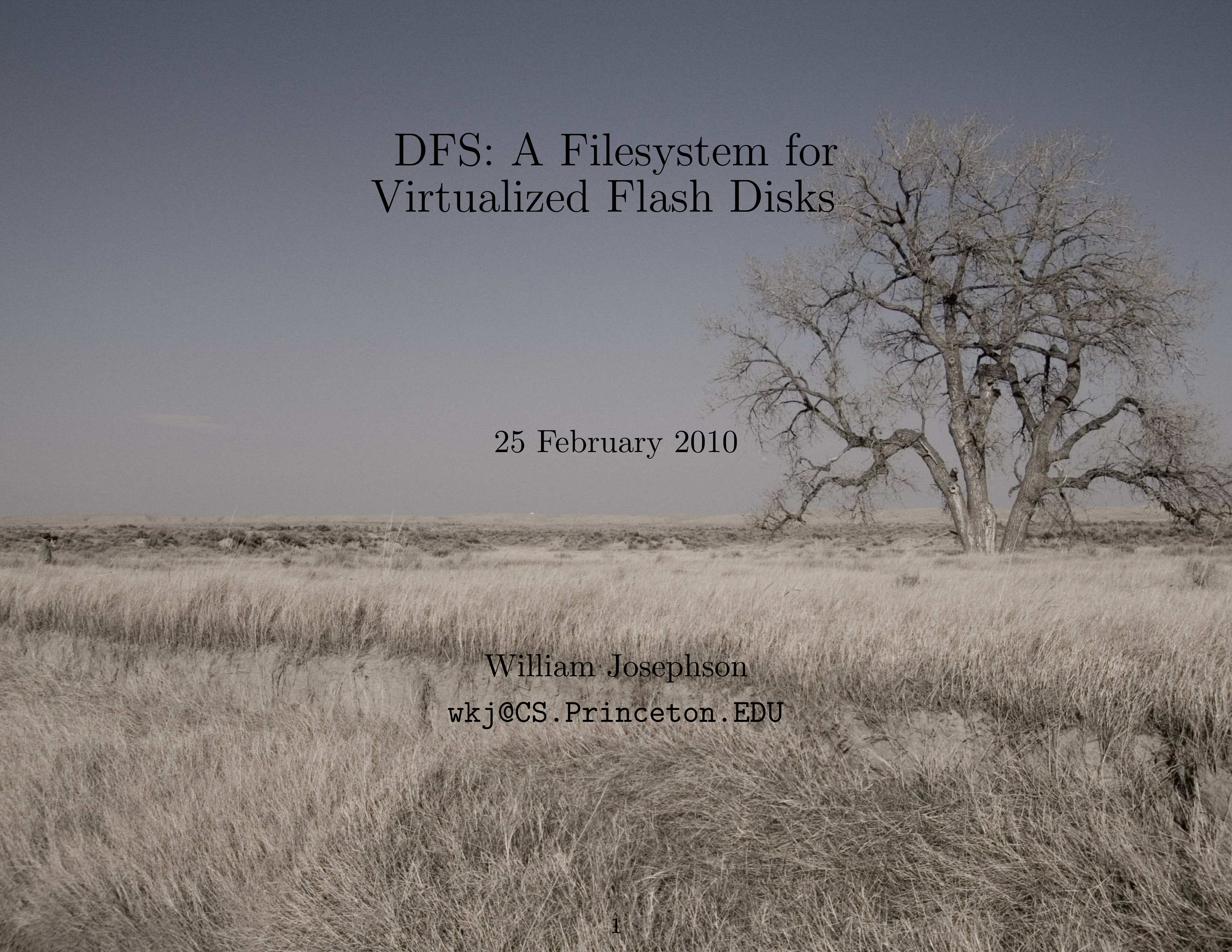


DFS: A Filesystem for Virtualized Flash Disks

25 February 2010

William Josephson
wkj@CS.Princeton.EDU

Why Flash?

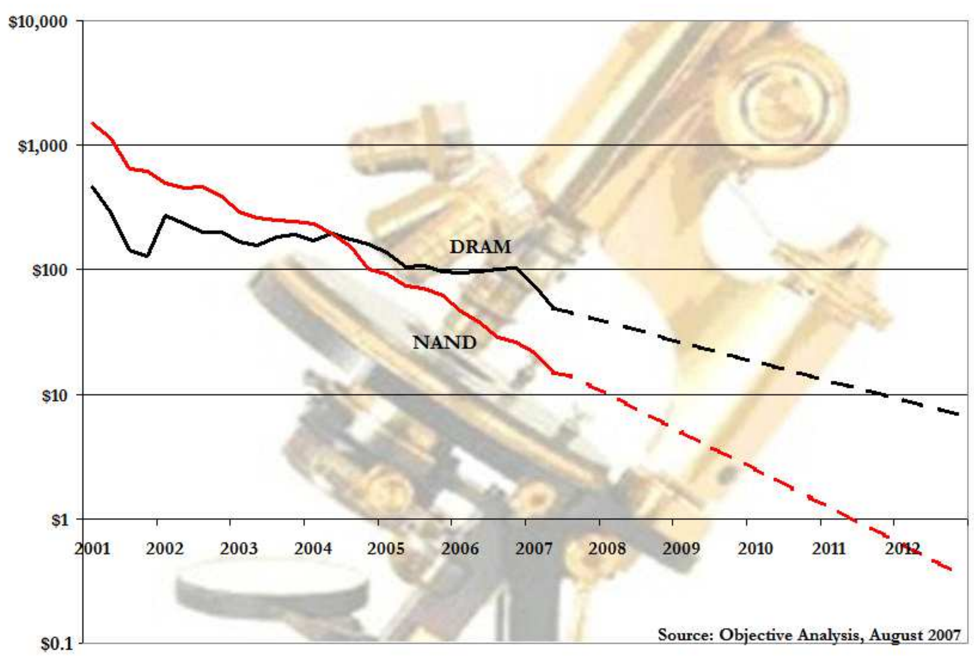
"Tape is Dead; Disk is Tape; Flash is Disk; RAM Locality is King"

-Jim Gray (2006)

- Why Flash?
  - Non-volatile storage
  - No mechanical components
    - Moore's law does not apply to seeks
  - Inexpensive and getting cheaper
  - Potential for significant power savings
  - Real-world performance is much better than in 2006
- **Bottom line**: disks for $/GB; flash for $/IOPS

Why not Battery-Backed DRAM?

- Flash costs less than DRAM and is getting cheaper
  - Both markets are volatile, however (*e.g.*, new iPhones)
- Memory subsystems that support large memory are expensive
- Think of flash as a new level in the memory hierarchy



- Last week's spot prices put SLC : DRAM at 1 : 3.6 and MLC at 1 : 9.8

Flash Memory Review

- Non-volatile solid state memory
  - Individual cells are comparable in size to a transistor
  - Not sensitive to mechanical shock
  - Re-write requires prior bulk erase
  - Limited number of erase/write cycles
- Two categories of flash:
  - NOR flash: random access, used for firmware
  - NAND flash: block access, used for mass storage
- Two types of memory cells:
  - SLC: single level cell that encodes a single bit per cell
  - MLC: multi-level cell that encodes multiple bits per cell

NAND Flash

- Economics
  - Individual cells are simple
    - Improved fabrication yield
    - 1st to use new process technology
  - Already must deal with failures, so just mark fab defects
  - High volume for many consumer applications
- Organization
  - Data is organized into "pages" for transfer (512B-4K)
  - Pages are grouped into "erase blocks" (EBs) (16K-16MB+)
  - Must erase an entire EB before writing again

NAND Flash Challenges

- Block oriented interface
  - Must read or write multiples of the page size
  - Must erase an entire EB at once
- Bulk erasure of EBs requires copying rather than update-in-place
- Limited number of erase cycles requires wear-leveling
  - Less of an issue if you are copying for performance anyway
- Additional error correction often necessary for reliability
- Performance requires HW parallelism and software support

Why Another Filesystem?

- There are many filesystems designed for spinning rust
  - *e.g.*, FFS, ext$N$, XFS, VxFS, FAT, NTFS, *etc.*
  - Layout not designed with flash in mind
  - Firmware/driver still implements a level of indirection
    - Indirection supports wear-leveling and copying for performance
- There are also several filesystems designed specifically for flash
  - *e.g.*, JFFS/JFFS2 (NOR), YAFFS/YAFFS2 (SLC NAND)
  - Log-structured; implement wear-leveling & additional ECC
  - Intended for embedded applications
  - Small numbers of files, small total filesystem sizes
  - Some must scan entire device at boot
  - Often expect to manage *raw* flash
- In a server environment, we end up with two storage managers!

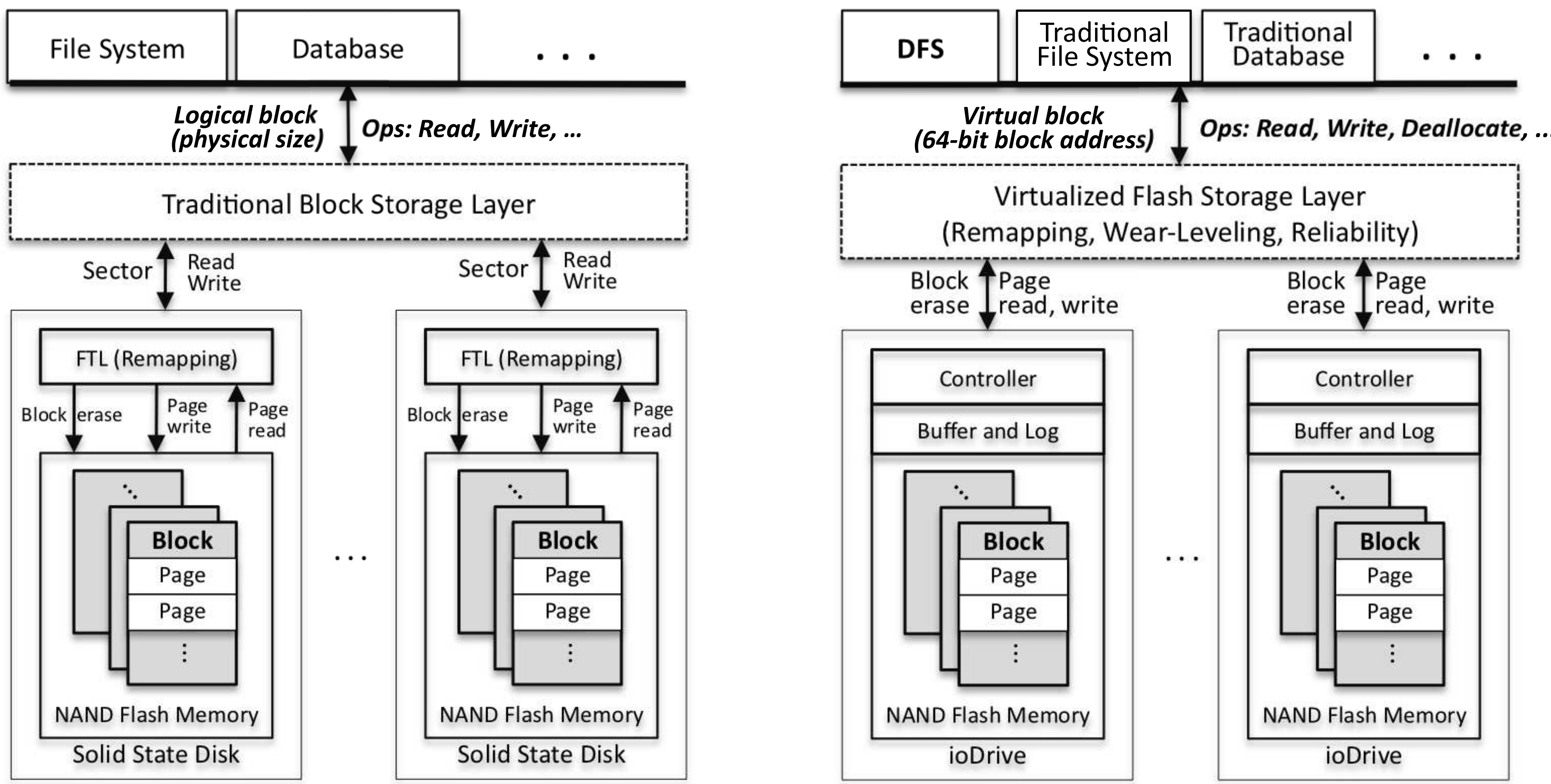
DFS: Idea

- Idea: Instead of running two storage managers, *delegate*
  - Filesystem still responsible for directory management, access control
  - Flash disk storage manager responsible for block allocation
  - May take advantage of features not in traditional disk interface
- Longer term question: what should storage interface look like?

DFS: Requirements

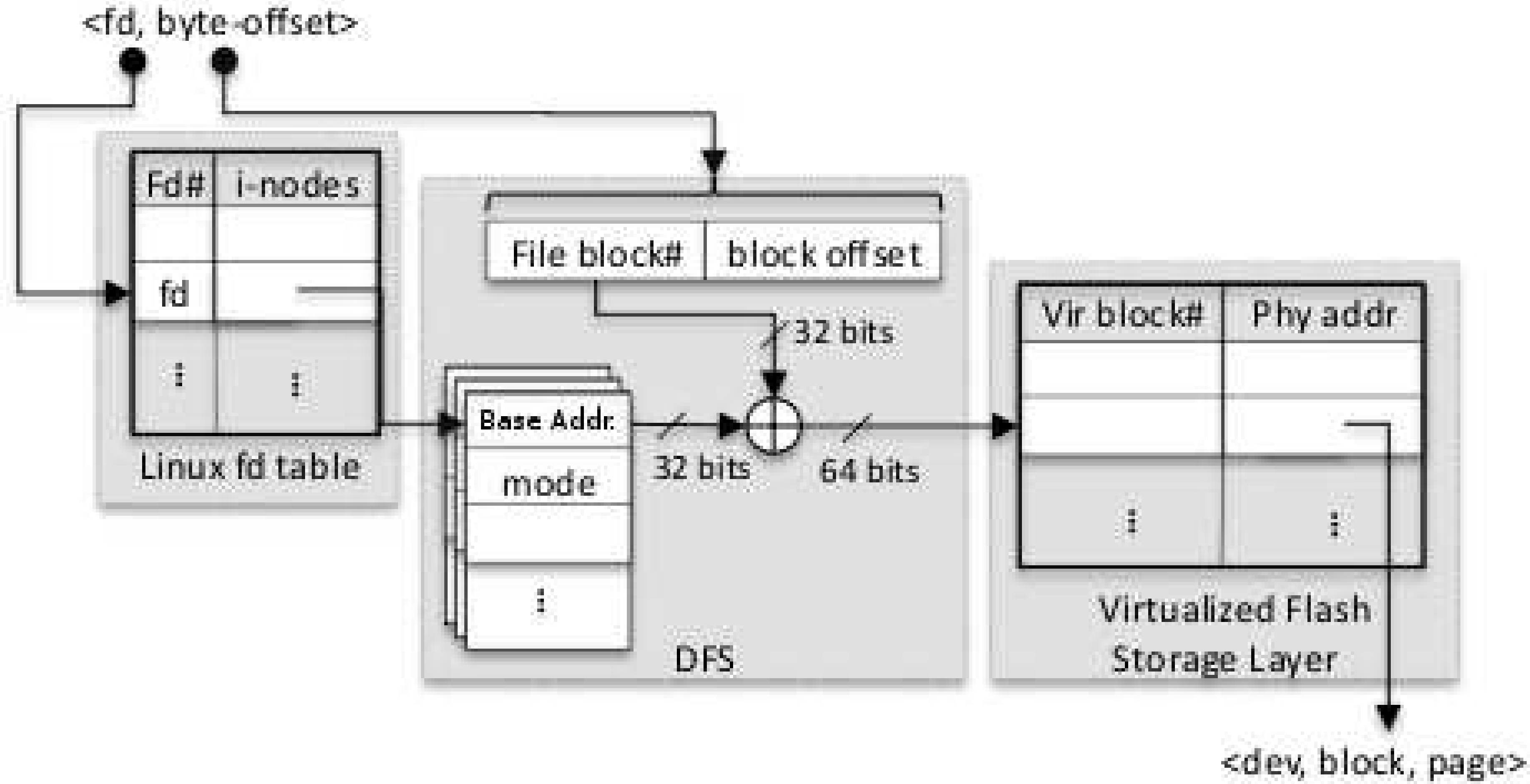
- Currently relies on four features of underlying flash disk
  1. Sparse block or object-based interface
  2. Crash recoverability of block allocations
  3. Atomic multi-block update
  4. Trim: *i.e.*, discard a block or block range
- All are a natural outgrowth of high-performance flash storage
  - (1) follows from block-remapping for copying and failed blocks
  - (2) and (3) follow from log-structured storage for write peformance
  - (4) already exists on most flash devices as a hint to GC

Block Diagram of Existing Approach *vs* DFS



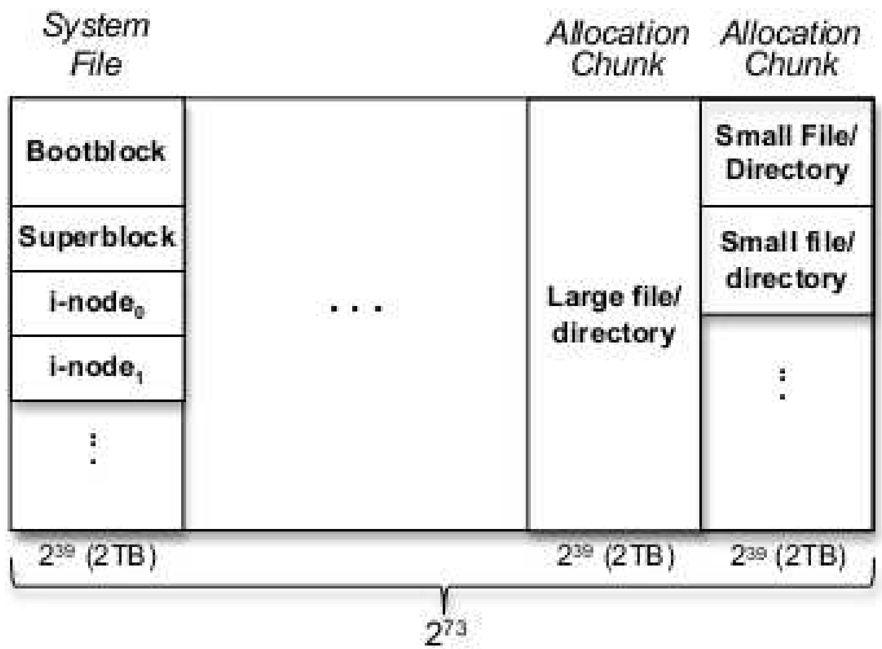
(a) Traditional layers of abstractions **(b) Our layers of abstractions**

DFS: Logical Address Translation



- I-node contains base virtual address for file's extent
- Base address, logical block #, and offset yield virtual address
- Flash storage manager translates virtual address to physical

DFS: File Layout



- Divide virtual address space into contiguous *allocation chunks*
  - Flash storage manager maintains sparse virtual-to-physical mapping
- First chunk used for boot block, super block, and I-nodes
- Subsequent chunks contain either one "large" file or several "small" files
- Size of allocation chunk and small file chosen at initialization

DFS: Directories

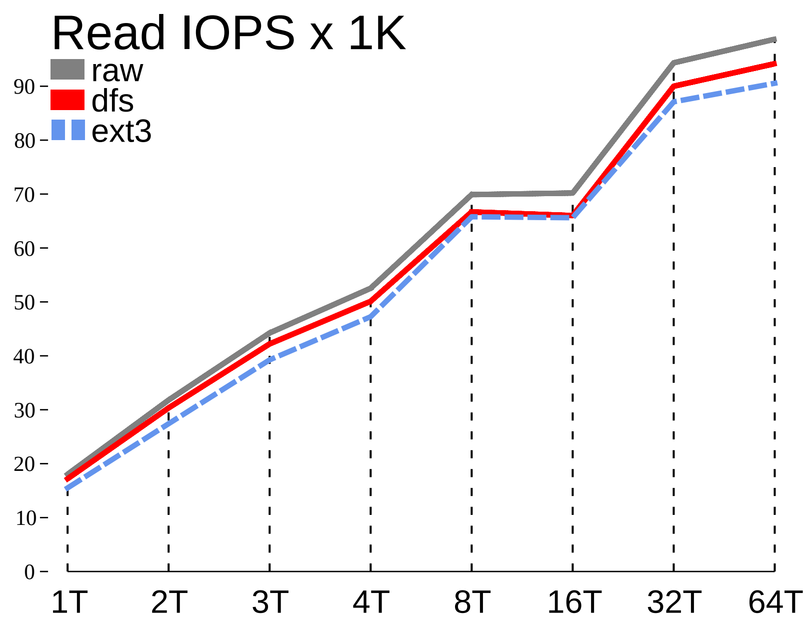
- Directory implementation that peforms is work in progress
  - Evaluation platform does not yet *export* atomic multi-block update
  - Plan to implement directories as sparse hash tables
- Current implementation uses UFS/FFS directory metadata
  - Requires additional logging of directory updates only

Evaluation Platform

- Linux 2.6.27.9 on a 4-core amd64 @ 2.4GHz with 4GB DRAM
- FusionIO ioDrive with 160GB SLC NAND flash (formatted capacity)
  - Sits on PCIe bus rather than SATA/SCSI bus
  - Hardware op latency is $\sim 50\mu s$
  - Theoretical peak throughput of $\sim 120,000$ IOPS
    * Version of device driver we are using limits throughput further
  - OS-specific device driver exports block device interface
    * Other features of the device can be separately exported
  - Functionality split between hardware, software, & host device driver
    * Device driver consumes host CPU and memory

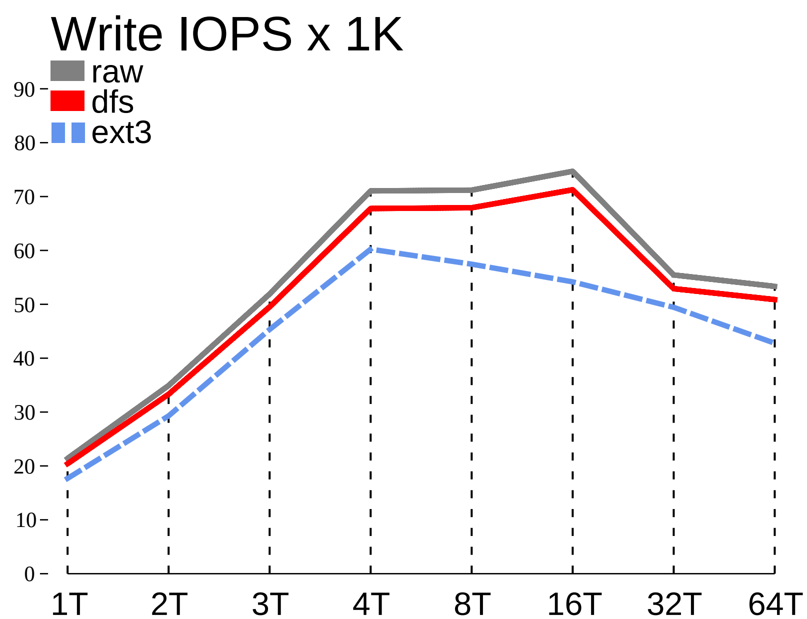
Microbenchmark: Random Reads

- Random 4KB I/Os per second as function of number of threads
  - Need multiple threads to take advantage of hardware parallelism
  - On our particular hardware, peak performance is about 100K IOPS
  - Host CPU/memory performance has substantial effect, too



Microbenchmark: Random Writes

- Random 4KB I/Os per second as function of number of threads
  - Once again need multiple threads to get best agregate performance
  - There is an additional garbage collector thread in device driver
- We consider CPU expended per I/O in a moment



Microbenchmark: CPU Utilization

- Improvement in CPU usage for DFS vs. Ext3 at peak throughput
  - *i.e.*, larger, positive number is better
- About the same for reads; improvement for writes at low concurrency
- 4 threads+4 cores: improved performance at higher cost due to GC

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

Application Benchmark: Description

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: Performance

Application	Wall Time: Ext3	Wall Time: DFS	Wall Time: 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

- Lower per-file lock contention
- I/Os to adjacent locations merged into fewer but larger requests
  - Simplified `get_block` can more easily issue contiguous I/O requests

Some Musings on Future Directions

- CPU overhead of device driver is not trivial
  - Particularly write side suffers from GC overhead
- Push storage management onto flash device or into network?
- No compelling reason to interact with flash as ordinary mass storage
  - Useful innovation at interface to new level in memory hierarchy?
    * Key/value pair interface implemented in hardware/firmware?
    * First class object store with additional metadata?

Conclusions

- With a little “secret sauce”, NAND flash becomes interesting
  - Secret sauce includes hardware, firmware, and possibly device driver
  - No need for flash to sit behind traditional mass storage bus
- Delegating storage management to flash vendor’s hardware/software:
  - Allows simplification of system software
  - Simulatenously provides opportunity for improved performance
  - Does not require changes to storage interfaces or protocols
    - There may be benefit to innovation in the storage interface
  - Allows vendors to improve the “secret sauce” independently

Acknowledgements

David Flynn at FusionIO

Garrett Swart at Oracle