DFS: A File System for Virtualized Flash Storage

William K. Josephson
wkj@CS.Princeton.EDU

Lars A. Bongo
larsab@Princeton.EDU

David Flynn
dflynn@FusionIO.COM

Kai Li
li@CS.Princeton.EDU

Abstract

This paper presents the design, implementation and evaluation of Direct File System (DFS) for virtualized flash storage. Instead of using traditional layers of abstraction, our layers of abstraction are designed for directly accessing flash memory devices. DFS has two main novel features. First, it lays out its files directly in a very large virtual storage address space provided by FusionIO's virtual flash storage layer. Second, it leverages the virtual flash storage layer to perform block allocations and atomic updates. As a result, DFS performs better and it is much simpler than a traditional Unix file system with similar functionalities. Our microbenchmark results show that DFS can deliver 94,000 I/O operations per second (IOPS) for direct reads and 71,000 IOPS for direct writes with the virtualized flash storage layer on FusionIO's ioDrive. For direct access performance, DFS is consistently better than ext3 on the same platform, sometimes by 20%. For buffered access performance, DFS is also consistently better than ext3, and sometimes by over 149%. Our application benchmarks show that DFS outperforms ext3 by 7% to 250% while requiring less CPU power.

1 Introduction

Flash memory has traditionally been the province of embedded and portable consumer devices. Recently, there has been significant interest in using it to run primary file systems for laptops as well as file servers in data centers. Compared with magnetic disk drives, flash can substantially improve reliability and random I/O performance while reducing power consumption. However, these file systems are originally designed for magnetic disks which may not be optimal for flash memory. A key systems design question is to understand how to build the entire system stack including the file system for flash memory.

Past research work has focused on building firmware and software to support traditional layers of abstractions for backward compatibility. For example, recently proposed techniques such as the flash translation layer (FTL) are typically implemented in a solid state disk controller with the disk drive abstraction [5, 6, 26, 3]. Systems software then uses a traditional block storage interface to support file systems and database systems designed and optimized for magnetic disk drives. Since flash memory is substantially different from magnetic disks, the rationale of our work is to study how to design new abstraction layers including a file system to exploit the potential of NAND flash memory.

This paper presents the design, implementation, and evaluation of the Direct File System (DFS) and describes the virtualized flash memory abstraction layer it uses for FusionIO's ioDrive hardware. The virtualized storage abstraction layer provides a very large, virtualized block addressed space, which can greatly simplify the design of a file system while providing backward compatibility with the traditional block storage interface. Instead of pushing the flash translation layer into disk controllers, this layer combines virtualization with intelligent translation and allocation strategies for hiding bulk erasure latencies and performing wear leveling.

DFS is designed to take advantage of the virtualized flash storage layer for simplicity and performance. A traditional file system is known to be complex and typically requires four or more years to become mature. The complexity is largely due to three factors: complex storage block allocation strategies, sophisticated buffer cache designs, and methods to make the file system crash-recoverable. DFS dramatically simplifies all three aspects. It uses virtualized storage spaces *directly* as a true single-level store and leverages the virtual to physical block allocations in the virtualized flash storage layer to avoid explicit file block allocations and reclamations. By doing so, DFS uses extremely simple metadata and data layout. As a result, DFS has a short datapath to flash memory and encourages users to access data directly instead of going through a large and complex buffer cache. DFS leverages the atomic update feature of the virtualized flash storage layer to achieve crash recovery.

We have implemented DFS for the FusionIO's virtualized flash storage layer and evaluated it with a suite of benchmarks. We have shown that DFS has two main advantages over the ext3 filesystem. First, our file system implementation is about one eighth that of ext3 with similar functionality. Second, DFS has much better performance than ext3 while using the same memory resources and less CPU. Our microbenchmark results show that DFS can deliver 94,000 I/O operations per second (IOPS) for direct reads and 71,000 IOPS direct writes with the virtualized flash storage layer on FusionIO's ioDrive. For direct access performance, DFS is consistently better than ext3 on the same platform, sometimes by 20%. For buffered access performance, DFS is also consistently better than ext3, and sometimes by over 149%. Our application benchmarks show that DFS outperforms ext3 by 7% to 250% while requiring less CPU power.

2 Background and Related Work

In order to present the details of our design, we first provide some background on flash memory and the challenges to using it in storage systems. We then provide an overview of related work.

2.1 NAND Flash Memory

Flash memory is a type of electrically erasable solid-state memory that has become the dominant technology for applications that require large amounts of non-volatile solid-state storage. These applications include music players, cell phones, digital cameras, and shock sensitive applications in the aerospace industry.

Flash memory consists of an array of individual cells, each of which is constructed from a single floating-gate transistor. Single Level Cell (SLC) flash stores a single bit per cell and is typically more robust; Multi-Level Cell (MLC) flash offers higher density and therefore lower cost per bit. Both forms support three operations: read, write (or program), and erase. In order to change the value stored in a flash cell it is necessary to perform an erase before writing new data. Read and write operations typically take tens of microseconds whereas the erase operation may take more than a millisecond.

The memory cells in a NAND flash device are arranged into pages which vary in size from 512 bytes to as much as 16KB each. Read and write operations are page-oriented. NAND flash pages are further organized into erase blocks, which range in size from tens of kilobytes to megabytes. Erase operations apply only to entire erase blocks; any data in an erase block that is to be preserved must be copied.

There are two main challenges in building storage systems using NAND flash. The first is that an erase operation typically takes about one or two milliseconds. The second is that an erase block may be erased successfully only a limited number of times. The endurance of an erase block depends upon a number of factors, but usually ranges from as little as 5,000 cycles for consumer grade MLC NAND flash to 100,000 or more cycles for enterprise grade SLC NAND flash.

2.2 Related Work

Douglis *et al.* studied the effects of using flash memory without a special software stack [11]. They showed that flash could improve read performance by an order of magnitude and decrease energy consumption by 90%, but that due to bulk erasure latency, write performance also decreased by a factor of ten. They further noted that large erasure block size causes unnecessary copies for cleaning, an effect often referred to as "write amplification".

Kawaguchi *et al.* [14] describe a transparent device driver that presents flash as a disk drive. The driver dynamically maps logical blocks to physical addresses, provides wear-leveling, and hides bulk erasure latencies using a log-structured approach similar to that of LFS [27]. State-of-the art implementations of this idea, typically called the Flash Translation Layer, have been implemented in the controllers of several high-performance Solid State Drives (SSDs) [3, 16].

More recent efforts focus on high-performance in SSDs, particularly for random writes. Birrell *et al.* [6], for instance, describe a design that significantly improves random write performance by keeping a fine-grained mapping between logical blocks and physical flash addresses in RAM. Similarly, Agrawal *et al.* [5] argue that SSD performance and longevity is strongly workload dependent and further that many systems problems that previously have appeared higher in the storage stack are now relevant to the device and its firmware. This observation has lead to the investigation of buffer management policies for a variety of workloads. Some policies, such as Clean First LRU (CFLRU) [24] trade off a reduced number of writes for additional reads. Others, such as Block Padding Least Recently Used (BPLRU) [15] are designed to improve performance for fine-grained updates or random writes.

eNVy [33] is an early file system design effort for flash memory. It uses flash memory as fast storage, a battery-backed SRAM module as a non-volatile cache for combining writes into the same flash block for performance, and copy-on-write page management to deal with bulk erasures

More recently, a number of file systems have been designed specifically for flash memory devices. YAFFS, JFFS2, and LogFS [19, 32] are example efforts that hide bulk erasure latencies and perform wear-leveling of NAND flash memory devices at the file system level using the log-structured approach. These file systems were initially designed for embedded applications instead of high-performance applications and are not generally suitable for use with the current generation of high-performance flash devices. For instance, YAFFS and JFFS2 manage raw NAND flash arrays directly. Furthermore, JFFS2 must scan the entire physical device at mount time which can take many minutes on large devices. All three filesystems are designed to access NAND flash chips directly, negating the performance advantages of the hardware and software in emerging flash device. LogFS does have some support for a block-device compatibility mode that can be used as a fall-back at the expense of performance, but none are designed to take advantage of emerging flash storage devices which perform their own flash management.

3 Our Approach

This section presents the three main aspects of our approach: (a) new layers of abstraction for flash memory storage systems which yield substantial benefits in simplicity and performance; (b) a virtualized flash storage layer, which provides a very large address space and implements dynamic mapping to hide bulk erasure latencies and to perform wear leveling; and (c) the design of DFS which takes full advantage of the virtualized flash storage layer. We further show that DFS is simple and performs better than the popular Linux ext3 file system.

3.1 Existing vs. New Abstraction Layers

Figure 1 shows the architecture block diagrams for existing flash storage systems and our proposed architecture. The traditional approach is to package flash memory as a solid-state disk (SSD) that exports a disk interface such as SATA or SCSI. An advanced SSD implements a flash translation layer (FTL) in its controller that maintains a dynamic mapping from logical blocks to physical flash pages to hide bulk erasure latencies and to perform wear leveling. Since a SSD uses the same interface as a magnetic disk drive, it supports the traditional block storage software layer which can be either a simple device driver or a sophisticated volume manager. The block storage layer then supports traditional file systems, database systems, and other software designed for magnetic disk drives. This approach has the advantage of disrupting neither the application-kernel interface nor the kernel-physical storage interface. On the other hand, it has a relatively thick software stack and makes it difficult for the software layers and hardware to take full advantage of the benefits of flash memory.

We advocate an architecture in which a greatly simplified file system is built on top of a virtualized flash storage layer implemented by the cooperation of the device driver and novel flash storage controller hardware. The controller exposes direct access to flash memory chips to the virtualized flash storage layer.

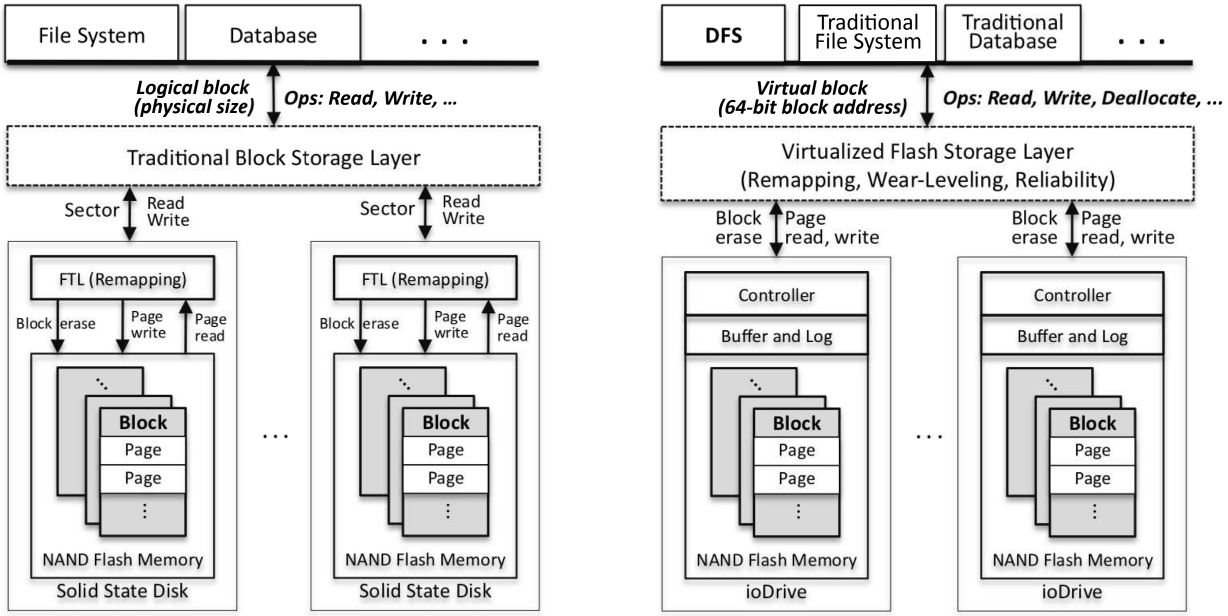
The virtualized flash storage layer is implemented at the device driver level which can freely cooperate with specific hardware support offered by the flash memory controller. The virtualized flash storage layer implements a large virtual block addressed space and maps it to physical flash pages. It handles multiple flash devices and uses a log-structured allocation strategy to hide bulk erasure latencies, perform wear leveling, and handle bad page recovery. This approach combines the virtualization and FTL together instead of pushing FTL into the disk controller layer. The virtualized flash storage layer can still provide backward compatibility to run existing file systems and database systems. The existing software can benefit from the intelligence in the device driver and hardware rather than having to implement that functionality independently in order to use flash memory. More importantly, flash devices are free to export a richer interface than that exposed by disk-based interfaces.

Direct File System (DFS) is designed to utilize the functionality provided by the virtualized flash storage layer. In addition to leveraging the support for wear-leveling and for hiding the latency of bulk erasures, DFS uses the virtualized flash storage layer to perform file block allocations and reclamations and uses atomic flash page updates for crash recovery. This architecture allows the virtualized flash storage layer to provide an object-based interface. Our main observation is that the separation of the file system from block allocations allows the storage hardware and block management algorithms to evolve jointly and independently from the file system and user-level applications. This approach makes it easier for the block management algorithms to take advantage of improvements in the underlying storage subsystem.

3.2 Virtualized Flash Storage Layer

The virtual flash storage layer provides an abstraction to enable client software such as file systems and database systems to take advantage of flash memory devices while providing backward compatibility with the traditional block storage interface. The primary novel feature of the virtualized flash storage layer is the provision for a very large, virtual block-addressed space. There are three reasons for this design. First, it provides client software with the flexibility to directly access flash memory in a single level store fashion across multiple flash memory devices. Second, it hides the details of the mapping from virtual to physical flash memory pages. Third, the flat virtual block-addressed space provides clients with a backward compatible block storage interface.

The mapping from virtual blocks to physical flash memory pages deals with several flash memory issues. Flash memory pages are dynamically allocated and reclaimed to hide the latency of bulk erasures, to distribute



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

Figure 1: Flash Storage Abstractions

writes evenly to physical pages for wear-leveling, and to detect and recover bad pages to achieve high reliability. Unlike a conventional Flash Translation Layer (FTL), the mapping supports a very large number of virtual pages – orders-of-magnitude larger than the available physical flash memory pages.

The virtualized flash storage layer currently supports three operations: read, write, and trim or deallocate. All operations are block-based operations, and the block size in the current implementation is 512 bytes. The write operation triggers a dynamic mapping from a virtual to physical page, thus there is no explicit allocation operation. The deallocate operation deallocates a range of virtual addresses. It removes the mappings of all mapped physical flash pages in the range and hands them to a garbage collector to recycle for future use. We anticipate that future versions of the VFSL will also support a move operation to allow data to be moved from one virtual address to another without incurring the cost of a read, write, and deallocate operation for each block to be copied.

The current implementation of the virtualized flash storage layer is a combination of a Linux device driver and FusionIO's ioDrive special purpose hardware. The ioDrive is a PCI Express card densely populated with either 160GB or 320GB of SLC NAND flash memory. The software for the virtualized flash storage layer is implemented as a device driver in the host operating system and leverages hardware support from the ioDrive itself.

The ioDrive uses a novel partitioning of the virtualized flash storage layer between the hardware and device driver to achieve high performance. The overarching design philosophy is to separate the data and control paths and to implement the control path in the device driver and the data path in hardware. The data path on the ioDrive card contains numerous individual flash memory packages arranged in parallel and connected to the host via PCI Express. As a consequence, the device achieves highest throughput with moderate parallelism in the I/O request stream. The use of PCI Express rather than an existing storage interface such as SCSI or SATA simplifies the partitioning of control and data paths between the hardware and device driver.

The device provides hardware support of checksum generation and checking to allow for the detection and correction of errors in case of the failure of individual flash chips. Metadata is stored on the device in terms of physical addresses rather than virtual addresses in order to simplify the hardware and allow greater throughput at lower economic cost. While individual flash pages are relatively small (512 bytes), erase blocks are several megabytes in size in order to amortize the cost of bulk erase operations.

The mapping between virtual and physical addresses is maintained by the kernel device driver. The mapping between 64-bit virtual addresses and physical addresses is maintained using a variation on B-trees in memory. Each address points to a 512-byte flash memory page, allowing a virtual address space of 2^{73} bytes. Updates are made stable by recording them in a log-structured fashion: the hardware interface is append-only. The device driver is also responsible for reclaiming unused storage using a garbage collection algorithm. Bulk erasure scheduling and wear leveling algorithms for flash endurance are integrated into the garbage collection component of the device driver.

A primary rationale for implementing the virtual to physical address translation and garbage collection in the device driver rather than in an embedded processor on the ioDrive itself is that the device driver can automatically take advantage of improvements in processor and memory bus performance on commodity hardware without requiring significant design work on a proprietary embedded platform. This approach does have the drawback of requiring potentially significant processor and memory resources on the host.

3.3 DFS

DFS is a full-fledged implementation of a Unix file system and it is designed to take advantage of several features of the virtualized flash storage layer, including large virtualized address space, direct flash access and its crash recovery mechanism. The implementation runs as a loadable kernel module in the Linux 2.6 kernel. The DFS kernel module implements the traditional Unix file system APIs via the Linux VFS layer. It supports the usual methods such as open, close, read, write, pread, pwrite, lseek, and mmap. The Linux kernel requires basic memory mapped I/O support in order to facilitate the execution of binaries residing on DFS file systems.

3.3.1 Leveraging Virtualized Flash Storage

DFS delegates I-node and file data block allocations and deallocations to the virtualized flash storage layer. The virtualized flash storage layer is responsible for block allocations and deallocations, for hiding the latency of bulk erasures, and for wear leveling.

We have considered two design alternatives. The first is to let the virtualized storage layer export an object-based interface. In this case, a separate object is used to represent each file system object and the virtualized flash storage layer is responsible for managing the underlying flash blocks. The main advantage of this approach is that it can provide a close match with what a file system implementation needs. The main disadvantage is the complexity of an object-based interface that provides backwards compatibility with the traditional block storage interface.

The second is to ask the virtualized flash storage layer to implement a large logical address space that is sparse. Each file system object will be assigned a contiguous range of logical block addresses. The main advantages of this approach are its simplicity and its natural support for the backward compatibility with the traditional block storage interface. The drawback of this approach is its potential waste of the virtual address space. DFS has taken this approach for its simplicity.

We have configured the ioDrive to export a sparse 64-bit logical block address space. Since each block contains 512 bytes, the logical address space spans 2^{73} bytes. DFS can then use this logical address space to map file system objects to physical storage.

DFS allocates virtual address space in contiguous "allocation chunks". The size of these chunks is configurable at file system initialization time but is 2^{32} blocks or 2TB by default. User files and directories are partitioned into two types: large and small. A large file occupies an entire chunk whereas multiple small files reside in a single chunk. When a small file grows to become a large file, it is moved to a freshly allocated chunk. The current implementation must implement this by copying the file contents, but we anticipate that future versions of the virtual flash storage layer will support changing the virtual to physical translation map without having to copy data. The current implementation does not support remapping large files into the small file range should a file shrink.

When the filesystem is initialized, two parameters must be chosen: the maximum size of a small file, which must be a power of two, and the size of allocation chunks, which is also the maximum size of a large file. These two parameters are fixed once the filesystem is initialized. They can be chosen in a principled manner given the anticipated workload. There have been many studies of file size distributions in different environments, for instance those by Tannenbaum *et al.* [28] and Docuer and Bolosky [10]. By default, small files are those less than 32KB.

The current DFS implementation uses a 32-bit I-node number to identify individual files and directories and a 32-bit block offset into a file. This means that DFS can support up to $-1 + 2^{32}$ files and directories in total since the first I-node number is reserved for the system. The largest supported file size is 2TB with 512-byte blocks since the block offset is 32 bits. The I-node itself stores the base virtual address for the logical extent containing the file data. This base address together with the file offset identifies the virtual address of a file block. Figure 2 depicts the mapping from file descriptor and offset to logical block address in DFS.

The very simple mapping from file and offset to logical block address has another beneficial implication. Each file is represented by a single logical extent, making it straightforward for DFS to combine multiple small I/O requests to adjacent regions into a single larger I/O. No complicated block layout policies are required at the filesystem layer. This strategy can improve performance because the flash device delivers higher transfer rates with larger I/Os. Our current implementation aggressively merges I/O requests; a more nuanced policy might improve performance further.

DFS leverages the three main operations supported by the virtualized flash storage layer: read from a logical block, write to a logical block, and discard a logical block range. The discard directive marks a logical block range

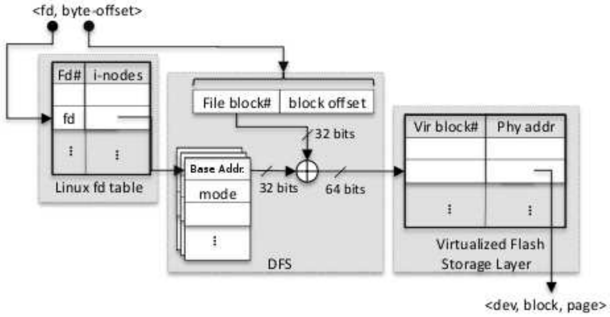


Figure 2: DFS logical block address mapping for large files; only the width of the file block number differs for small files

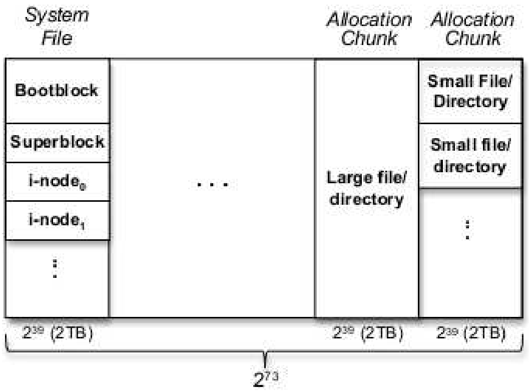


Figure 3: Layout of DFS system and user files in virtualized flash storage. The first 2TB is used for system files. The remaining 2TB allocation chunks are for user data or directory files. A large file takes the whole chunk; multiple small files are packed into a single chunk.

as garbage for the garbage collector and ensures that subsequent reads to the range return only zeros. A version of the discard directive already exists in many flash devices as a hint to the garbage collector; DFS, by contrast, depends upon it to implement truncate and remove. It is also possible to interrogate a logical block range to determine if it contains allocated blocks. The current version of DFS does not make use of this feature, but it could be used by archival programs such as `tar` that have special representations for sparse files.

3.3.2 DFS Layout and Objects

The DFS file system uses a simple approach to store files and their metadata. It divides the 64-bit block addressed virtual flash storage space (DFS volume) into block addressed subspaces or allocation chunks. The size of these two types of subspaces are configured when the filesystem is initialized. DFS places large files in their own allocation chunks and stores multiple small files in a chunk.

As shown in Figure 3, there are three kinds of files in the DFS file system. The first file is a system file which includes the boot block, superblock and all I-nodes. This file is a "large" file and occupies the first allocation chunk at the beginning of the raw device. The boot block occupies the first few blocks (sectors) of the raw device. A superblock immediately follows the boot block. At mount time, the file system can compute the location of the superblock directly. The remainder of the system file contains all I-nodes as an array of block-aligned I-node data structures.

Each I-node is identified by a 32-bit unique identifier or I-node number. Given the I-node number, the logical address of the I-node within the I-node file can be computed directly. Each I-node data structure is stored in a single 512-byte flash block. Each I-node contains the I-number, base virtual address of the corresponding file, mode, link count, file size, user and group IDs, any special flags, a generation count, and access, change, birth, and modification times with nanosecond resolution. These fields take a total of 72 bytes, leaving 440 bytes for additional attributes and future use. Since an I-node fits in a single flash page, it will be updated atomically by the virtualized flash storage layer.

The implementation of DFS uses a 32-bit block-addressed allocation chunk to store the content of a regular file. Since a file is stored in a contiguous, flat space, the address of each block offset can be simply computed by adding the offset to the virtual base address of the space for the file. A block read simply returns the content of the physical flash page mapped to the virtual block. A write operation writes the block to the mapped physical flash page directly. Since the virtualized flash storage layer triggers a mapping or remapping on write, DFS does the write without performing an explicit block allocation. Note that DFS allows holes in a file without using physical flash pages because of the dynamic mapping. When a file is deleted, the DFS will issue a deallocation operation provided by the virtualized flash storage layer to deallocate and unmap virtual space of the entire file.

A DFS directory is mapped to flash storage in the same manner as ordinary files. The only difference is its internal structure. A directory contains contains an array of name, I-node number, type triples. The current implementation is very similar to that found in FFS [22]. Updates to directories, including operations such as rename, which touch multiple directories and the on-flash I-node allocator, are made crash-recoverable through the use of a write-ahead log. Although widely used and simple to implement, this approach does not scale well to large directories. The current version of the virtualized flash storage layer does not export atomic multi-block updates. We anticipate reimplementing directories using hashing and a sparse virtual address space made crash recoverable with atomic updates.

3.3.3 Direct Data Accesses

DFS promotes direct data access. The current Linux implementation of DFS allows the use of the buffer cache in order to support memory mapped I/O which is required for the `exec` system call. However, for many workloads of interest, particularly databases, clients are expected to bypass the buffer cache altogether. The current implementation of DFS provides direct access via the direct I/O buffer cache bypass mechanism already present in the Linux kernel. Using direct I/O, page-aligned reads and writes are converted directly into I/O requests to the block device driver by the kernel.

There are two main rationales for this approach. First, traditional buffer cache design has several drawbacks. The traditional buffer cache typically uses a large amount of memory. Buffer cache design is quite complex since it needs to deal with multiple clients, implement sophisticated cache replacement policies to accommodate various access patterns of different workloads, and maintain consistency between the buffer cache and disk drives, and support crash recovery. In addition, having a buffer cache imposes a memory copy in the storage software stack.

Second, flash memory devices provide low-latency accesses, especially for random reads. Since the virtualized flash storage layer can solve the write latency problem, the main motivation for the buffer cache is largely eliminated. Thus, applications can benefit from the DFS direct data access approach by utilizing most of the main memory space typically used for the buffer cache for a larger in memory working set.

3.3.4 Crash Recovery

The virtualized flash storage layer implements the basic functionality of crash recovery for the mapping from logical block addresses to physical flash storage locations. DFS leverages this property to provide crash recovery. Unlike traditional file systems that use non-volatile random access memory (NVRAM) and their own logging implementation, DFS piggybacks on the flash storage layer's log.

NVRAM and file system level logging require complex implementations and introduce additional costs for the traditional file systems. NVRAM is typically used in high-end file systems so that the file system can achieve low-latency operations while providing fault isolations and avoiding data loss in case of power failures. The traditional logging approach is to log every write and performs group commits to reduce overhead. Logging writes to disk can impose significant overheads. A more efficient approach is to log updates to NVRAM, which is the method typically used in high-end file systems [12]. NVRAMs are typically implemented with battery-backed DRAMs on a PCI card whose price is similar to a few high-density magnetic disk drives. NVRAMs can substantially reduce the file system write performance because every write must go through the NVRAM. For a network file system, each write will have to go through the I/O bus three times, once for the NIC, once for NVRAM, and once for writing to disks.

Since flash memory is a form of NVRAM, DFS leverages the support from the virtualized flash storage layer to achieve crash recoverability. When a DFS file system object is extended, DFS passes the write request to the virtualized flash storage layer which then allocates a physical page of the flash device and logs the result internally. After a crash, the virtualized flash storage layer runs recovery using the internal log. The consistency of the contents of individual files is the responsibility of applications, but the on-flash state of the file system is guaranteed to be consistent. Since the virtualized flash storage layer uses a log-structured approach to tracking allocations for performance reasons and must handle crashes in any case, DFS does not impose any additional onerous requirements.

3.3.5 Discussion

The current DFS implementation has several limitations. The first is that it does not yet support snapshots. One of the reasons we did not implement snapshot is that we plan to support snapshots natively in the virtualized flash storage layer which will greatly simplify the snapshot implementation in DFS. Since the virtualized flash storage layer is already log-structured for performance and hence takes a copy-on-write approach by default, one can implement snapshots in the virtualized flash storage layer efficiently.

The second is that we are currently implementing support for atomic multi-block updates in the virtualized flash storage layer. The log-structured, copy-on-write nature of the flash storage layer makes it possible to export such an interface efficiently. For example, Prabhakaran *et al.* recently described an efficient commit protocol to implement atomic multi-block writes [25]. This type of methods will allow DFS to guarantee the consistency of directory contents and I-node allocations in a simple fashion. In the interim, DFS uses a straightforward extension of the traditional UFS/FFS directory structure.

The third is the limitations on the number of files and the maximum file size. We have considered a design that supports two file sizes: small and very large. The file layout algorithm initially assumes a file is small (*e.g.,* less than 2GB). If it needs to exceed the limit, it will become a very large file (*e.g.,* up to 2PB). The virtual block address space is partitioned so that a large number of small file ranges are mapped in one partition and a smaller number of very large file ranges are mapped into the remaining partition. A file may be promoted from the small partition to the very large partition by copying the mapping of a virtual flash storage address space to another at the virtualized flash storage layer. We plan to export such support and implement this design in the next version of DFS.

4 Evaluation

We are interested in answering two main questions:

- How do the layers of abstraction perform?
- How does DFS compare with existing file systems?

To answer the first question, we use a microbenchmark to evaluate the number of I/O operations per second (IOPS) and bandwidth delivered by the virtualized flash storage layer and by the DFS layer. To answer the second question, we compare DFS with ext3 by using a microbenchmark and an application suite. Ideally, we would compare with existing flash filesystems as well, however filesystems such as YAFFS and JFFS2 are designed to use raw NAND flash and are not compatible with next-generation flash storage that exports a block interface.

All of our experiments were conducted on a desktop with Intel Quad Core processor running at 2.4GHz with a 4MB cache and 4GB DRAM. The host operating system was a stock Fedora Core installation running the Linux 2.6.27.9 kernel. Both DFS and the virtualized flash storage layer implemented by the FusionIO device driver were compiled as loadable kernel modules.

We used a FusionIO ioDrive with 160GB of SLC NAND flash connected via PCI-Express x4 [1]. The advertised read latency of the FusionIO device is $50\mu s$. For a single reader, this translates to a theoretical maximum throughput of 20,000 IOPS. Multiple readers can take advantage of the hardware parallelism in the device to achieve much higher aggregate throughput. For the sake of comparison, we also ran the microbenchmarks on a 32GB Intel X25-E SSD connected to a SATA II host bus adapter [2]. This device has an advertised typical read latency of about $75\mu s$.

Our results show that the virtualized flash storage layer delivers performance close to the limits of the hardware, both in terms of IOPS and bandwidth. Our results also show that DFS is much simpler than ext3 and achieves better performance in both the micro- and application benchmarks than ext3, often using less CPU power.

4.1 Virtualized Flash Storage Performance

We have two goals in evaluating the performance of the virtualized flash storage layer. First, to examine the potential benefits of the proposed abstraction layer in combination with hardware support that exposes parallelism. Second, to determine the raw performance in terms of bandwidth and IOPs delivered in order to compare DFS and ext3. For both purposes, we designed a simple microbenchmark which opens the raw block device in direct I/O mode, bypassing the kernel buffer cache. Each thread in the program attempts to execute block-aligned reads and writes as quickly as possible.

To evaluate the benefits of the virtualized flash storage layer and its hardware, one would need to compare a traditional block storage software layer with flash memory hardware equivalent to the FusionIO ioDrive but with a traditional disk interface FTL. Since such hardware does not exist, we have used a Linux block storage layer with an Intel X25-E SSD, which is a well-regarded SSD in the marketplace. Although this is not a fair comparison, the results give us some sense of the performance impact of the abstractions designed for flash memory.

We measured the number of sustained random I/O transactions per second. While both flash devices are enterprise class devices, the test platform is the typical white box workstation we described earlier. The results are shown in Figure 4. Performance, while impressive compared to magnetic disks, is less than that advertised by the manufacturers. We suspect that the large IOPS performance gaps, particularly for write IOPS, are partially limited by the disk drive interface and limited resources in a drive controller to run sophisticated remapping algorithms.

Device	Read IOPS	Write IOPS
Intel	33,400	3,120
FusionIO	98,800	75,100

Figure 4: Device 4KB Peak Random IOPS

Device	Threads	Read (MB/s)	Write (MB/s)
Intel	2	221	162
FusionIO	2	769	686

Figure 5: Device Peak Bandwidth 1MB Transfers

Figure 5 shows the peak bandwidth for both cases. We measured sequential I/O bandwidth by computing the aggregate throughput of multiple readers and writers. Each client transferred 1MB blocks for the throughput test and used direct I/O to bypass the kernel buffer cache. The results in the table are the bandwidth results using two writers. The virtualized flash storage layer with ioDrive achieves 769MB/s for read and 686MB/s for write, whereas the traditional block storage layer with the Intel SSD achieves 221MB/s for read and 162MB/s for write.

4.2 Complexity of DFS vs. ext3

Figure 6 shows the number of lines of code for the major modules of DFS and ext3 file systems. Although both implement Unix file systems, DFS is much simpler. The

Module	DFS	Ext3
Headers	392	1583
Kernel Interface (Superblock, *etc.*)	1625	2973
Logging	0	7128
Block Allocator	0	1909
I-nodes	250	6544
Files	286	283
Directories	561	670
ACLs, Extended Attrs.	N/A	2420
Resizing	N/A	1085
Miscellaneous	175	113
Total	3289	24708

Figure 6: Lines of Code in DFS and Ext3 by Module

simplicity of DFS is mainly due to delegating block allocations and reclamations to the virtualized flash storage layer. The ext3 file system, for example, has a total of 17,500 lines of code and relies on an additional 7,000 lines of code to implement logging (JBD) for a total of nearly 25,000 lines of code compared to roughly 3,300 lines of code in DFS. Of the total lines in ext3, about 8,000 lines (33%) are related to block allocations, deallocations and I-node layout. Of the remainder, another 3,500 lines (15%) implement support for on-line resizing and extended attributes, neither of which are supported by DFS.

Although it may not be fair to compare a research prototype file system with a file system that has evolved for several years, the percentages of block allocation and logging in the file systems give us some indication of the relative complexity of different components in a file system.

4.3 Microbenchmark Performance of DFS vs. ext3

We use Iozone [23] to evaluate the performance of DFS and ext3 on the ioDrive when using both direct and buffered access. We record the number of 4KB I/O transactions per second achieved with each file system and also compute the CPU usage required in each case as the ratio between user plus system time to elapsed wall time. For both file systems, we ran Iozone in three different modes: in the default mode in which I/O requests pass through the kernel buffer cache, in direct I/O mode without the buffer cache, and in memory-mapped mode using the `mmap` system call.

In our experiments, both file systems run on top of the virtualized flash storage layer. The ext3 file system in this case uses the backward compatible block storage interface supported by the virtualized flash storage layer.

Direct Access

For both reads and writes, we consider sequential and uniform random access to previously allocated blocks. Our goal is to understand the additional overhead due to DFS compared to the virtualized flash storage layer. The results indicate that DFS is indeed lightweight and imposes much less overhead than ext3. Compared to the raw device, DFS delivers about 5% fewer IOPS for both read and write whereas ext3 delivers 9% fewer read IOPS and more than 20% fewer write IOPS. In terms of bandwidth, DFS delivers about 3% less write bandwidth whereas ext3 delivers 9% less write bandwidth.

File System	Threads	Read (MB/s)	Write (MB/s)
ext3	2	760	626
DFS	2	769	667

Figure 7: Peak Bandwidth 1MB Transfers on ioDrive

Figure 7 shows the peak bandwidth for sequential 1MB block transfers. This microbenchmark is the filesystem analog of the raw device bandwidth performance shown in Figure 5. Although the performance difference between DFS and ext3 for large block transfers is relatively modest, DFS does narrow the gap between filesystem and raw device performance for both sequential reads and writes.

Figure 8 shows the average direct random I/O performance on DFS and ext3 as a function of the number of concurrent clients on the FusionIO ioDrive. Both of the file systems also exhibit a characteristic that may at first seem surprising: *aggregate* performance often increases with an increasing number of clients, even if the client requests are independent and distributed uniformly at random. This behavior is due to the relatively long latency of individual I/O transactions and deep hardware and software request queues in the flash storage subsystem. This behavior is quite different from what most applications expect and may require changes to them in order to realize the full potential of the storage system.

Unlike read throughput, write throughput peaks at about 16 concurrent writers and then decreases slightly. Both the aggregate throughput and the number of concurrent writers at peak performance are lower than when accessing the raw storage device. The additional overhead imposed by the filesystem on the write path reduces both the total aggregate performance and the number of concurrent writers that can be handled efficiently.

We have also measured CPU utilization per 1,000 IOPS delivered in the microbenchmarks. Figure 9 shows the improvement of DFS over ext3. We report the average of five runs of the IOZone based microbenchmark with a standard deviation of one to three percent. For reads, DFS CPU utilization is comparable to ext3; for writes, particularly with small numbers of threads, DFS is more efficient. Overall, DFS consumes somewhat less CPU power, further confirming that DFS is a lighter weight file system than ext3.

One anomaly worthy of note is that DFS is actually

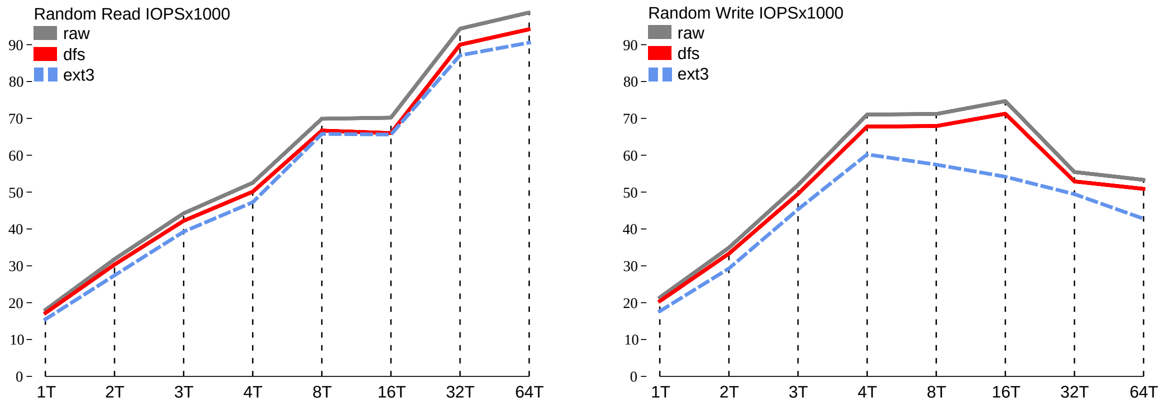


Figure 8: Aggregate IOPS for 4K Random Direct I/O as a Function of the Number of Threads

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

Figure 9: Improvement in CPU Utilization per $1,000$ IOPS using 4K Direct I/O with DFS relative to Ext3

more expensive than ext3 per I/O when running with four clients, particularly if the clients are writers. This is due to the fact that there are four cores on the test machine and the device driver itself has worker threads that require CPU and memory bandwidth. The higher performance of DFS translates into more work for the device driver and particularly for the garbage collector. Since there are more threads than cores, cache hit rates suffer and scheduling costs increase; under higher offered load, the effect is more pronounced, although it can be mitigated somewhat by binding the garbage collector to a single processor core.

Buffered Access

To evaluate the performance of DFS in the presence of the kernel buffer cache, we ran a similar set of experiments as in the case of direct I/O. Each experiment touched 8GB worth of data using 4K block transfers. The buffer cache was invalidated after each run by unmounting the file system and the total data referenced exceeded the physical memory available by a factor of two. The first run of each experiment was discarded and the average of the subsequent ten runs reported.

Figures 10 and 11 show the results via the Linux buffer cache and via memory-mapped I/O data path which also uses the buffer cache. There are several observations.

	Seq. Read IOPS x 1K		Rand. Read IOPS x 1K	
Thr.	ext3	DFS (Speedup)	ext3	DFS (Speedup)
1	125.5	191.2 (1.52)	17.5	19.0 (1.09)
2	147.6	194.1 (1.32)	32.9	34.0 (1.03)
3	137.1	192.7 (1.41)	44.3	46.6 (1.05)
4	133.6	193.9 (1.45)	55.2	57.8 (1.05)
8	134.4	193.5 (1.44)	78.7	80.5 (1.02)
16	132.6	193.9 (1.46)	79.6	81.1 (1.02)
32	132.3	194.8 (1.47)	95.4	101.2 (1.06)
	Seq. Write IOPS x 1K		Rand. Write IOPS x 1K	
Thr.	ext3	DFS (Speedup)	ext3	DFS (Speedup)
1	67.8	154.9 (2.28)	61.2	68.5 (1.12)
2	71.6	165.6 (2.31)	56.7	64.6 (1.14)
3	73.0	156.9 (2.15)	59.6	62.8 (1.05)
4	65.5	161.5 (2.47)	57.5	63.3 (1.10)
8	64.9	148.1 (2.28)	57.0	58.7 (1.03)
16	65.3	147.8 (2.26)	52.6	56.5 (1.07)
32	65.3	150.1 (2.30)	55.2	50.6 (0.92)

Figure 10: Buffer Cache Performance with 4KB I/Os

First, both DFS and ext3 have similar random read IOPS and random write IOPS to their performance results using direct I/O. Although this is expected, DFS is better than ext3 on average by about 5%. This further shows that DFS has less overhead than ext3 in the presence of a buffer cache.

Second, we observe that the traditional buffer cache is not effective when there are a lot of parallel accesses. In the sequential read case, the number of IOPS delivered by DFS basically doubles its direct I/O access performance, whereas the IOPS of ext3 is only modestly better than its random access performance when there are enough parallel accesses. For example, when there are 32 threads, its IOPS is 132,000, which is only 28% better than its random read IOPS of 95,400!

Third, DFS is substantially better than ext3 for both sequential read and sequential write cases. For sequential reads, it outperforms ext3 by more than a factor of 1.4. For sequential writes, it outperforms ext3 by more than a

	Seq. Read IOPS x 1K		Rand. Read IOPS x 1K	
Thr.	ext3	DFS (Speedup)	ext3	DFS (Speedup)
1	42.6	52.2 (1.23)	13.9	18.1 (1.3)
2	72.6	84.6 (1.17)	22.2	28.2 (1.27)
3	94.7	114.9 (1.21)	27.4	32.1 (1.17)
4	110.2	117.1 (1.06)	29.7	35.0 (1.18)
	Seq. Write IOPS x 1K		Rand. Write IOPS x 1K	
Thr.	ext3	DFS (Speedup)	ext3	DFS (Speedup)
1	28.8	40.2 (1.4)	11.8	13.5 (1.14)
2	39.9	55.5 (1.4)	16.7	18.1 (1.08)
3	41.9	68.4 (1.6)	19.1	20.0 (1.05)
4	44.3	70.8 (1.6)	20.1	22.0 (1.09)

Figure 11: Memory Mapped Performance of Ext3 & DFS

factor of 2.15. This is largely due to the fact that DFS is simple and can easily combines I/Os.

The story for memory-mapped I/O performance is much the same as it is for buffered I/O. Random access performance is relatively poor compared to direct I/O performance. The simplicity of DFS and the short code paths in the filesystem allow it to outperform ext3 in this case. The comparatively large speedups for sequential I/O, particularly sequential writes, is again due to the fact that DFS readily combines multiple small I/Os into larger ones. In the next section we show that I/O combining is an important effect; the quicksort benchmark is a good example of this phenomenon with memory mapped I/O. We count both the number of I/O transactions during the course of execution and the total number of bytes transferred. DFS greatly reduces the number of write operations and more modestly the number of read operations.

4.4 Application Benchmarks Performance of DFS vs. ext3

We have used five applications as an application benchmark suite to evaluate the application-level performance on DFS and ext3.

Application Benchmarks

The table in Figure 12 summarizes the characteristics of the applications and the reasons why they are chosen for our performance evaluation.

In the following, we describe each application, its implementation and workloads in detail:

Quicksort. This quicksort is implemented as a single-threaded program to sort 715 million 24 byte key-value pairs memory mapped from a single 16GB file. Although quicksort exhibits good locality of reference, this benchmark program nonetheless stresses the memory mapped I/O subsystem. The memory-mapped interface has the advantages of being simple, easy to understand, and a straightforward way to transform a large flash storage device into an inexpensive replacement for DRAM as it provides the illusion of word-addressable access.

Applications	Description	I/O Patterns
Quicksort	A quicksort on a large dataset	Mem-mapped I/O
N-Gram	A program for querying n-gram data	Direct, random read
KNNImpute	Processes bioinformatics microarray data	Mem-mapped I/O
VM-Update	Update of an OS on several virtual machines	Sequential read & write
TPC-H	Standard benchmark for Decision Support	Mostly sequential read

Figure 12: Applications and their characteristics.

N-Gram. This program indexes all of the 5-grams in the Google n-gram corpus by building a single large hash table that contains 26GB worth of key-value pairs. The Google n-gram corpus is a large set of n-grams and their appearance counts taken from a crawl of the Web that has proved valuable for a variety of computational linguistics tasks. There are just over 13.5 million words or 1-grams and just over 1.1 billion 5 grams. Indexing the data set with an SQL database takes a week on a computer with only 4GB of DRAM [9]. Our indexing program uses 4KB buckets with the first 64 bytes reserved for metadata. The implementation does not support overflows, rather an occupancy histogram is constructed to find the smallest k such that 2^k hash buckets will hold the dataset without overflows. With a variant of the standard Fowler-Nolls-Vo hash, the entire data set fits in 16M buckets and the histogram in 64MB of memory. Our evaluation program uses synthetically generated query traces of 200K queries each; results are based upon the average of twenty runs. Queries are drawn either uniformly at random or according to a Zipf distribution with $\alpha = 1.0001$. The results were qualitatively similar for other values of α until locking overhead dominated I/O overhead.

KNNImpute. This program is a very popular bionformatics code for estimating missing values in data obtained from microarray experiments. The program uses the KNNImpute [29] algorithm for DNA microarrays which takes as input a matrix with G rows representing genes and E columns representing experiments. Then a symmetric GxG distance matrix with the Euclidean distance between all gene pairs is calculated based on all experiment values for both genes. Finally, the distance matrix is written to disk as its output. The program is a multi-threaded implementation using memory-mapped I/O. Our input data is a matrix with 41,768 genes and 200 experiments results in a matrix of about 32MB, and a distance matrix of 6.6GB. There are 2079 genes with missing values.

VM Update. This benchmark is a simple update of multiple virtual machines hosted on a single server. We

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

Figure 13: Application Benchmark Execution Time Improvement: Best of DFS *vs* Best of Ext3

choose this application because virtual machines have become popular from both a cost and management perspective. Since each virtual machine typically runs the same operating system but has its own copy, operating system updates can pose a significant performance problem. Each virtual machine needs to apply critical and periodic system software updates at the same time. This process is both CPU and I/O intensive. To simulate such an environment, we installed 4 copies of Ubuntu 8.04 in four different VirtualBox instances. In each image, we downloaded all of the available updates and then measured the amount of time it took to install these updates. There were a total of 265 packages updated containing 343MB of compressed data and about 38,000 distinct files.

TPC-H. This is a standard benchmark for decision support workloads. We used the Ingres database to run the Transaction Processing Council's Benchmark H (TPC-H) [4]. The benchmark consists of 22 business oriented queries and two functions that respectively insert and delete rows in the database. We used the default configuration for the database with two storage devices: the database itself, temporary files, and backup transaction log were placed on the flash device and the executables and log files were stored on the local disk. We report the results of running TPC-H with a scale factor of 5, which corresponds to about 5GB of raw input data and 90GB for the data, indexes, and logs stored on flash once loaded into the database.

Performance Results of DFS vs. ext3

This section first reports the performance results of DFS and ext3 for each application, and then analyzes the results in detail.

The main performance result is that DFS improves applications substantially over ext3. Figures 13 shows the elapsed wall time of each application running with ext3 and DFS on the same execution environment mentioned at the beginning of the section. The results show that DFS improves the performance all applications and the speedups range from a factor of 1.07 to 2.47.

To explain the performance results, we will first use Figure 14 to show the number of read and write IOPS, and the number of bytes transferred for reads and writes for each application. The main observation is that DFS issues a smaller number of larger I/O transactions than ext3, though the behaviors of reads and writes are quite different. This observation explains partially why DFS improves the performance of all applications, since we know from the microbenchmark performance that DFS achieves better IOPS than ext3 and significantly better throughput when the I/O transaction sizes are large.

One reason for larger I/O transactions is that in the Linux kernel, file offsets are mapped to block numbers via a per-file-system `get_block` function. The DFS implementation of `get_block` is aggressive about making large transfers when possible. A more nuanced policy might improve performance further, particularly in the case of applications such as KNNImpute and the VM Update workload which actually see an increase in the total number of bytes transferred. In most cases, however, the result of the current implementation is a modest reduction in the number of bytes transferred.

But, the smaller number of larger I/O transactions does not completely explain the performance results. In the following, we will describe our understanding of the performance of each application individually.

Quicksort. The Quicksort benchmark program sees a speedup of 1.54 when using DFS instead of ext3 on the ioDrive. Unlike the other benchmark applications, the quicksort program sees a large increase in CPU utilization when using DFS instead of ext3. CPU utilization includes both the CPU used by the FusionIO device driver and by the application itself. When running on ext3, this benchmark program is I/O bound; the higher throughput provided by DFS leads to higher CPU utilization, which is actually a desirable outcome in this particular case. In addition, we collected statistics from the virtualized flash storage layer to count the number of read and write transactions issued in each of the three cases. When running on ext3, the number of read transactions is similar to that found with DFS, whereas the number of write transactions is roughly twenty-five times larger than that of DFS, which contributed to the speedup. The average transaction size with ext3 is about 4KB instead of 64KB with DFS.

Google N-Gram Corpus. The N-gram query benchmark program running on DFS achieves a speedup of 2.5 over that on ext3. Figure 15 illustrates the speedup as a function of the number of concurrent threads; in all cases, the internal cache is 1,024 hash buckets and all I/O bypasses the kernel's buffer cache.

The hash table implementation is able to achieve about 95% of the random I/O performance delivered in the `Iozone` microbenchmarks given sufficient concurrency. As expected, performance is higher when the queries are Zipf-distributed as the internal cache captures many of the most popular queries. For Zipf parameter $\alpha = 1.0001$, there are about 156,000 4K random reads to sat-

	Read IOPS x 1000		Read Bytes x 1M		Write IOPS x 1000		Write Bytes x 1M	
Application	Ext3	DFS (Change)	Ext3	DFS (Change)	Ext3	DFS (Change)	Ext3	DFS (Change)
Quick Sort	1989	1558 (0.78)	114614	103991 (0.91)	49576	1914 (0.04)	203063	192557 (0.95)
N-Gram (Zipf)	156	157 (1.01)	641	646 (1.01)	N/A	N/A	N/A	N/A
KNNImpute	2387	1916 (0.80)	42806	36146 (0.84)	2686	179 (0.07)	11002	12696 (1.15)
VM Update	244	193 (0.79)	9930	9760 (0.98)	3712	1144 (0.31)	15205	19767 (1.30)
TPC-H	6375	3760 (0.59)	541060	484985 (0.90)	52310	3626 (0.07)	214265	212223 (0.99)

Figure 14: App. Benchmark Improvement in IOPS Required and Bytes Transferred: Best of DFS *vs* Best of Ext3

	Wall Time in Sec.		Ctx Switch x 1K	
Threads	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

Figure 15: Zipf-Distributed N-Gram Queries: Elapsed Time and Context Switches ($\alpha = 1.0001$)

isfy 200,000 queries. Moreover, query performance for hash tables backed by DFS scales with the number of concurrent threads much as it did in the `Iozone` random read benchmark. The performance of hash tables backed by ext3 do not scale with the number of threads nearly so well. This is due to increased per-file lock contention in ext3. We measured the number of voluntary context switches when running on each file system as reported by `getrusage`. A voluntary context switch indicates that the application was unable to acquire a resource in the kernel such as a lock. When running on ext3, the number of voluntary context switches increased dramatically with the number of concurrent threads; it did not do so on DFS. Although it may be possible to overcome the resource contention in ext3, the simplicity of DFS allows us to sidestep the issue altogether. This effect was less pronounced in the microbenchmarks because `Iozone` never assigns more than one thread to each file by default.

Bioinformatics Missing Value Estimation. KNNImpute takes about 18% less time to run when using DFS as opposed to ext3 with a standard deviation of about 1% of the mean run time. About 36% of the total execution time when running on ext3 is devoted to writing the distance matrix to stable storage. Most of the improvement in run time when running on DFS is during this phase of execution. CPU utilization increases by almost 7% on average when using DFS instead of ext3. This is due to increased system CPU usage during the distance matrix write phase by the FusionIO device driver's worker threads, particularly the garbage collector.

Virtual Machine Update. On average, it took 648 seconds to upgrade virtual machines hosted on DFS and 701 seconds to upgrade those hosted on ext3 file systems, for a net speed up of 7.6%. In both cases, the four virtual machines used nearly all of the available CPU for the duration of the benchmark. We found that each VirtualBox instance kept a single processor busy almost 25% percent of the time even when the guest operating system was idle. As a result, the virtual machine update workload quickly became CPU bound. If the virtual machine implementation itself were more efficient or more virtual machines shared the same storage system we would expect to see a larger benefit to using DFS.

TPC-H. We ran the TPC-H benchmark with a scale factor of five on both DFS and ext3. The average speedup over five runs was 1.22. For the individual queries DFS always performs better than ext3, with the speedup ranging from 1.04 (Q1: pricing summary report) to 1.51 (RF2: old sales refresh function). However, the largest contribution to the overall speedup is the 1.20 speedup achieved for Q5 (local supplier volume), which consumes roughly 75% of the total execution time.

There is a large reduction (14.4x) in the number of write transactions when using DFS as compared to ext3 and a smaller reduction (1.7x) in the number of read transactions. As in the case of several of the other benchmark applications, the large reduction in the number of I/O transactions is largely offset by larger transfers in each transaction, resulting in a modest decrease in the total number of bytes transferred.

CPU utilization is lower when running on DFS as opposed to ext3, but the Ingres database thread runs with close to 100% CPU utilization in both cases. The reduction in CPU usage is due instead to greater efficiency in the kernel storage software stack, particularly the flash device driver's worker threads.

5 Conclusion

This paper presents the design, implementation, and evaluation of DFS and describes FusionIO's virtualized flash storage layer. We have demonstrated that novel layers of abstraction specifically for flash memory can yield substantial benefits in software simplicity and system performance.

We have learned several things from DFS design. First, DFS is simple and has a short and direct way to access flash memory. Much of its simplicity comes from leveraging the virtualized flash storage layer such as large virtual storage space, block allocation and deallocation, and atomic block updates.

Second, the simplicity of DFS translates into performance. Our microbenchmark results show that DFS can deliver 94,000 IOPS for random reads and 71,000 IOPS random writes with the virtualized flash storage layer on FusionIO's ioDrive. The performance is close to the hardware limit.

Third, DFS is substantially faster than ext3. For direct access performance, DFS is consistently faster than ext3 on the same platform, sometimes by 20%. For buffered access performance, DFS is also consistently faster than ext3, and sometimes by over 149%. Our application benchmarks show that DFS outperforms ext3 by 7% to 250% while requiring less CPU power.

We have also observed that the impact of the traditional buffer cache diminishes when using flash memory. When there are 32 threads, the sequential read throughput of DFS is about twice that for direct random reads with DFS, whereas ext3 achieves only a 28% improvement over direct random reads with ext3.

References

[1] FusionIO ioDrive specification sheet. `http://www.fusionio.com/PDFs/Fusion%20Specsheet.pdf`.

[2] Intel X25-E SATA solid state drive. `http://download.intel.com/design/flash/nand/extreme/extreme-sata-ssd-datasheet.pdf`.

[3] Understanding the flash translation layer (FTL) specification. Tech. Rep. AP-684, Intel Corporation, December 1998.

[4] *TPC Benchmark H: Decision Support*. Transaction Processing Performance Council (TPC), 2008. `http://www.tpc.org/tpch`.

[5] AGRAWAL, N., PRABHAKARAN, V., WOBBER, T., DAVIS, J. D., MANASSE, M., AND PANIGRAHY, R. Design tradeoffs for SSD performance. In *Proceedings of the 2008 USENIX Technical Conference* (June 2008).

[6] BIRRELL, A., ISARD, M., THACKER, C., AND WOBBER, T. A design for high-performance flash disks. *ACM Operating Systems Review 41*, 2 (April 2007).

[7] BRANTS, T., AND FRANZ, A. Web 1T 5-gram version 1, 2006.

[8] CARD, R., T'SO, T., AND TWEEDIE, S. The design and implementation of the second extended filesystem. In *First Dutch International Symposium on Linux* (December 1994).

[9] CARLSON, A., MITCHELL, T. M., AND FETTE, I. Data analysis project: Leveraging massive textual corpora using n-gram statistics. Tech. Rep. CMU-ML-08-107, Carnegie Mellon University Machine Learning Department, May 2008.

[10] DOUCEUR, J. R., AND BOLOSKY, W. J. A large scale study of file-system contents. In *Proceedings of the 1999 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (1999).

[11] DOUGLIS, F., CACERES, R., KAASHOEK, M. F., LI, K., MARSH, B., AND TAUBER, J. A. Storage alternatives for mobile computers. In *Operating Systems Design and Implementation* (1994), pp. 25–37.

[12] HITZ, D., LAU, J., AND MALCOM, M. File system design for an nfs file server appliance. Tech. Rep. TR-3002, NetApp Corporation, September 2001.

[13] JO, H., KANG, J.-U., PARK, S.-Y., KIM, J.-S., AND LEE, K. FAB: Flash-aware buffer management policy for portable media players. *IEEE Transactions on Consumer Electronics 52*, 2 (2006), 485–493.

[14] KAWAGUCHI, A., NISHIOKA, S., AND MOTODA, H. A flash-memory based file system. In *In Proceedings of the Winter 1995 USENIX Technical Conference* (1995).

[15] KIM, H., AND AHN, S. BPLRU: A buffer management scheme for improving random writes in flash storage. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies* (February 2008).

[16] KIM, J., KIM, J. M., NOH, S. H., MIN, S. L., , AND CHO, Y. A space-efficient flash translation layer for CompactFlash systems. *IEEE Transactions on Consumer Electronics 48*, 2 (2002), 366–375.

[17] LI, K. Towards a low power file system. Tech. Rep. CSD-94-814, University of California at Berkeley, May 1994.

[18] LLANOS, D. R. TPCC-UVa: An open-source TPC-C implementation for global performance measurement of comptuer systems. *ACM SIGMOD Record 35*, 4 (December 2006), 6–15.

[19] MANNING, C. YAFFS: The NAND-specific flash file system. *LinuxDevices.Org* (September 2002).

[20] MARSH, B., DOUGLIS, F., AND KRISHNAN, P. Flash memory file caching for mobile computers. In *Proceedings of the Twenty-Seventh Hawaii Internanal Conference on Architecture* (January 1994).

[21] MATHUR, A., CAO, M., BHATTACHARYA, S., DILGER, A., TOMAS, A., AND VIVIER, L. The new ext4 filesystem: Current status and future plans. In *Ottowa Linux Symposium* (June 2007).

[22] MCKUSICK, M. K., JOY, W. N., LEFFLER, S. J., AND FABRY, R. S. A fast file system for UNIX. *ACM Transactions on Computer Systems 2*, 3 (August 1984).

[23] NORCOTT, W. Iozone filesystem benchmark. `http://www.iozone.org`.

[24] PARK, S.-Y., JUNG, D., KANG, J.-U., KIM, J.-S., AND LEE, J. CFLRU: A replacement algorithm for flash memory. In *Procieedings of the 2006 International Conference on Compilers, Architecture and Syntehsis for embedded Systems* (2006).

[25] PRABHAKARAN, V., RODEHEFFER, T. L., AND ZHOU, L. Transactional flash. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation* (December 2008).

[26] RAJIMWALE, A., PRABHAKARAN, V., AND DAVIS, J. D. Block management in solid state devices. Unpublished Technical Report, January 2009.

[27] ROSENBLUM, M., AND OUSTERHOUT, J. K. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems 10* (1992), 1–15.

[28] TANENBAUM, A. S., HERDER, J. N., AND BOS, H. File size distribution in UNIX systems: Then and now. *ACMSIGOPS Operating Systems Review 40*, 1 (January 2006), 100–104.

[29] TROYANSKAYA, O., CANTOR, M., SHERLOCK, G., BROWN, P., HASTIEEVOR, T., TIBSHIRANI, R., BOTSTEIN, D., AND ALTMAN, R. B. Missing value estimation methods for DNA microarrays. *Bioinformatics 17*, 6 (2001), 520–525.

[30] TWEEDIE, S. Ext3, journaling filesystem. In *Ottowa Linux Symposium* (July 2000).

[31] ULMER, C., AND GOKHALE, M. Threading opportunities in high-performance flash-memory storage. In *High Performance Embedded Computing* (2008).

[32] WOODHOUSE, D. JFFS: The journalling flash file system. In *Ottowa Linux Symposium* (2001).

[33] WU, M., AND ZWAENEPOEL, W. eNVy: A non-volatile, main memory storage system. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems* (1994).