WILLIAM K. JOSEPHSON, LARS A. BONGO, DAVID FLYNN, AND KAI LI

DFS: a file system for virtualized flash storage

William Josephson is in the PhD program at Princeton University, where he works with Kai Li. His research interests include high-performance storage and systems support for search in large-scale, high-dimensional data sets.

wkj@CS.Princeton.EDU

Lars Ailo Bongo is a post-doctoral researcher at the Lewis-Sigler Institute for Integrative Genomics at Princeton University. He received his PhD at the University of Tromsø.His research interests include system support for bioinformatics applications.

lbongo@Princeton.EDU

As President and CTO of Fusion-io and one of the company's founders, David Flynn is the visionary behind Fusion-io's innovative technology. Mr. Flynn is responsible for providing business-focused oversight of the company's research and development efforts, as well as driving the company's short- and long-term technological direction.

dflynn@FusionIO.COM

Kai Li is a Paul M. Wythes and Marcia R. Wythes Professor in the computer science department of Princeton University, with research interests in operating systems, parallel and distributed systems, storage systems, and analyzing and visualizing large datasets. He co-founded Data Domain, Inc., which pioneered deduplication storage systems.

li@CS.Princeton.EDU

WHILE FLASH MEMORY HAS TRADITIONALLY been the province of embedded and portable consumer devices, there has been recent interest in using flash devices to run primary file systems for laptops as well as file servers. Compared with magnetic disk drives, flash can substantially improve reliability and random I/O performance while reducing power consumption. However, file systems originally designed for magnetic disks are not optimal for flash memory. In this article we examine a flash device used as a disk replacement and how a file system that delegates block allocation to the device driver outperforms the ext3 [15] file system when used with the same device.

Past research work has focused on building firmware and software to support the traditional layers of abstractions used in file systems. For example, techniques such as the flash translation layer (FTL) are typically implemented in a solid-state disk controller that exports a traditional disk drive abstraction [3, 5, 6, 12]. 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 has very different performance characteristics from magnetic disks (there is no seek or rotation latency), we wanted to study and design new abstraction layers, including a file system to exploit the potential of next-generation NAND flash storage devices.

We describe the design and implementation of the Direct File System (DFS) and the virtualized flash memory (storage) 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 FTL into disk controllers, this layer combines virtualization with intelligent translation and allocation strategies for hiding the bulk erasure latencies and performing wear leveling required by flash memory devices.

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 uses virtualized storage *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 an extremely simple metadata and data layout. As a result, DFS has a short data path to flash memory and encourages users to access data directly instead of going through a large and complex buffer cache. DFS also leverages the atomic update feature of the virtualized flash storage layer to achieve crash recoverability.

We have implemented DFS for the FusionIO's virtualized flash storage layer and evaluated it with a suite of benchmarks [9]. We have shown that DFS has two main advantages over the ext3 file system. First, our file system implementation is about one-eighth the size of that of ext3, with similar functionality. Second, DFS has much better performance than ext3, while using the same memory and less CPU. Our micro-benchmark 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, sometimes by over 149%. Our application benchmarks show that DFS outperforms ext3 by 7% to 250%, while requiring fewer CPU resources.

NAND Flash

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. Flash memory consists of an array of individual cells, each of which is constructed from a single floating-gate transistor. Flash cells 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; erase operations only apply to entire erase blocks, and any data 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.

Our Approach

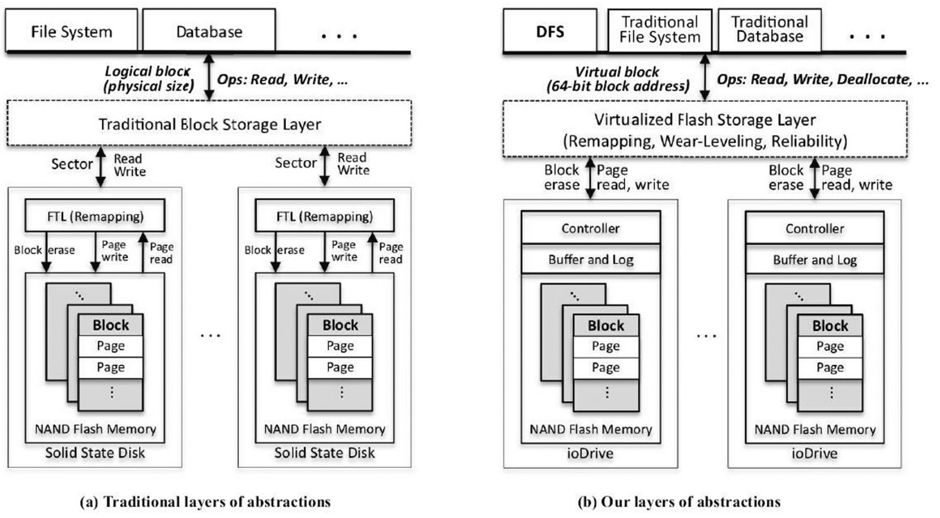
File System
Database
Logical block (physical size)
Ops: Read, Write, ...
Traditional Block Storage Layer
Sector
Read Write
FTL (Remapping)
Block erase
Page write
Page read
Block
Page
NAND Flash Memory
Solid State Disk
(a) Traditional layers of abstractions
DFS
Traditional File System
Traditional Database
Virtual block (64-bit block address)
Ops: Read, Write, Deallocate, ...
Virtualized Flash Storage Layer (Remapping, Wear-Leveling, Reliability)
Block erase
Page read, write
Controller
Buffer and Log
ioDrive
(b) Our layers of abstractions

FIGURE 1: FLASH STORAGE ABSTRACTIONS

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 the flash translation layer in its controller and maintains a dynamic mapping from logical blocks to physical flash pages to hide bulk erasure latencies and to perform wear leveling. SSDs use the same electrical and software interfaces as magnetic disk drives. The block storage layer above the disk interface supports traditional file systems, database systems, and other software. This approach has the advantage of not disrupting the application-kernel or kernel-physical storage interfaces. 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, which is implemented at the device driver level and 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.

The virtualized flash storage layer can still provide backward compatibility to run existing file systems and database systems. Existing software can benefit from the intelligence in the device driver and hardware. 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. 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.

VIRTUALIZED FLASH STORAGE LAYER

The virtual flash storage layer provides an abstraction that allows client software such as file systems and database systems to take advantage of flash memory devices while providing a backward-compatible 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 familiar block 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 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, 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 a physical page; thus, there is no explicit allocation operation. The deallocate operation deallocates a range of virtual addresses and notifies the garbage collector.

The current implementation of the virtualized flash storage layer is a combination of a closed source Linux device driver and FusionIO's ioDrive special-purpose hardware. The ioDrive is a PCI Express card 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 the 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.

DFS

DFS is a full-fledged implementation of a UNIX file system that is designed to take advantage of the virtualized flash storage layer. 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 execute binaries residing on DFS file systems.

LEVERAGING VIRTUALIZED FLASH STORAGE

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 delegates I-node and file data block allocations and deallocations to the virtualized flash storage layer.

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 size of these allocation chunks and the maximum size of small files can be chosen in a principled manner when the file system is initialized. There have been many studies of file size distributions in different environments (e.g., Tanenbaum et al. [13], Douceur and Bolosky [8]). By default, small files are those less than 32KB.

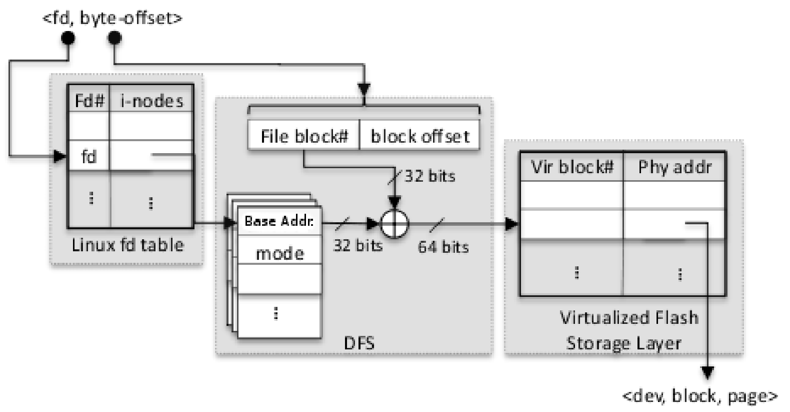

FIGURE 2: DFS LOGICAL BLOCK ADDRESS MAPPING FOR LARGE FILES. ONLY THE WIDTH OF THE FILE BLOCK NUMBER DIFFERS FOR SMALL FILES.

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 a total of up to $2^{32} - 1$ files and directories (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 the added benefit of making it straightforward for DFS to combine multiple small I/O requests to adjacent regions of a file into a single larger I/O. This strategy can improve performance, because the flash device delivers higher transfer rates with larger I/Os.

DFS LAYOUT AND OBJECTS

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. The remainder of the system file contains all I-nodes as an array of block-aligned I-node data structures.

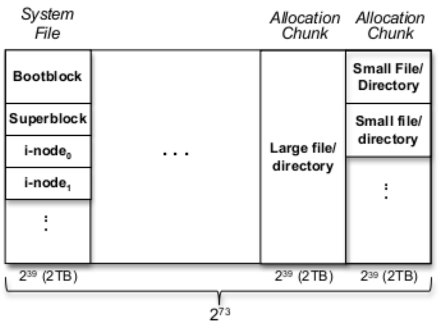

FIGURE 3: LAYOUT OF DFS SYSTEM AND USER FILES IN VIRTUALIZED FLASH STORAGE. THE FIRST 2TB ARE USED FOR SYSTEM FILES. THE REMAINING 2TB ALLOCATION CHUNKS ARE FOR USER DATA OR DIRECTORY FILES. A LARGE FILE TAKES THE WHOLE CHUNK; MULTIPLE SMALL FILES ARE PACKED INTO A SINGLE CHUNK.

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 re-mapping 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 the 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 an array of name, I-node number, and type triples. The current implementation is very similar to that found in FFS [11]. 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.

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 by the kernel directly into I/O requests to the block device driver.

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, 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.

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.

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.

DISCUSSION

The current DFS implementation has several limitations. The first is that it does not yet support snapshots. 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. In the interim, DFS uses a straightforward extension of the traditional UFS/FFS directory structure. The third is the limitation on the number and on the maximum size of files.

Evaluation

Application	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 4: APPLICATIONS AND THEIR CHARACTERISTICS

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 micro-benchmark 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 micro-benchmark and an application suite. Ideally, we would compare with existing flash file systems as well; however, file systems such as YAFFS [10] and JFFS2 [16] are designed to use raw NAND flash and are not compatible with the FusionIO hardware.

	Wall Time		
Application	*Ext3*	*DFS*	*Speedup*
Quicksort	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 5: APPLICATION BENCHMARK EXECUTION TIME IMPROVEMENT: BEST OF DFS VS. BEST OF EXT3

All of our experiments were conducted on a desktop with an 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 50μs. 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 micro-benchmarks 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 75μs.

We have evaluated our design and implementation with both a collection of micro-benchmarks and an application benchmark suite. Figure 4 summarizes the applications in the benchmark and their characteristic I/O request patterns. Figure 5 shows the elapsed wall time for each of the applications for both ext3 and DFS and the speedup, which varies from 1.07 to 2.47.

The quicksort application is a single-threaded sort of 715 million 24-byte key-value pairs memory mapped from a single 16GB file that is four times larger than main memory. Although quicksort exhibits good locality of reference, this benchmark program nonetheless stresses the memory-mapped I/O subsystem.

The n-gram benchmark issues random queries against a single large hash table index of the 5-grams in the Google n-gram corpus [7], which contains a large set of n-grams and their appearance counts taken from a crawl of the Web. The resulting index, which contains 26GB worth of small key-value pairs for 5-grams alone, has proved valuable for a variety of computational linguistics tasks. We present the results for a Zipf-distributed query distribution over the 5-grams.

The KNNImpute [14] benchmark program is a very popular bioinformatics code for estimating missing values in data obtains from wet lab microarray experiments. The program is a multi-threaded implementation using memory-mapped I/O.

The virtual machine update benchmark consists of a full operating system update of several VirtualBox instances running Ubuntu 8.04 hosted on a single server. Since each virtual machine typically runs the same operating system but has its own copy, operating system updates can pose a significant performance problem in some environments, as each instance needs to apply critical and periodic system software updates simultaneously. In our benchmark environment there were a total of 265 packages updated, containing 343MB of compressed data and about 38,000 distinct files.

The last benchmark program is the standard Transaction Processing Council's Benchmark H (TPC-H) [2]. We used the Ingres database to run the benchmark at scale factor 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.

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. Our paper includes the results of several additional benchmarks, including micro-benchmarks. These results were excluded from this article due to space constraints.

Conclusion

This article 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 the DFS design process. First, it is possible to implement DFS so that it is both simple and has short, direct-path flash memory. Much of its simplicity comes from leveraging the virtualized flash storage layer for large virtual storage space, block allocation and deallocation, and atomic block updates.

Second, the simplicity of DFS translates into performance. Our micro-benchmark 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 of 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/products/iodrive/.

[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: Technical report AP-684, Intel Corporation, December 1998.

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

[5] N. Agrawal, V. Prabhakaran, T. Wobber, J.D. Davis, M. Manasse, and R. Panigrahy, "Design Tradeoffs for SSD Performance," *Proceedings of the 2008 USENIX Annual Technical Conference* (USENIX Association, 2008).

[6] A. Birrell, M. Isard, C. Thacker, and T. Wobber, "A Design for High-Performance Flash Disks," *ACM Operating Systems Review*, vol. 41, no. 2 (April 2007).

[7] T. Brants and A. Franz, Web 1T 5-gram Version 1, 2006.

[8] J.R. Douceur and W.J. Bolosky, "A Large Scale Study of File-System Contents," *Proceedings of the 1999 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (1999).

[9] W. Josephson, L. Bongo, D. Flynn, and K. Li, "DFS: A File System for Virtualized Flash Storage," *Proceedings of FAST '10: 8th USENIX Conference on File and Storage Technologies* (USENIX Association, 2010), pp. 85–100.

[10] C. Manning, "YAFFS: The NAND-Specific Flash File System," *LinuxDevices.Org,* September 2002.

[11] M.K. McKusick, W.N. Joy, S.J. Leffler, and R.S. Fabry, "A Fast File System for UNIX," *ACM Transactions on Computer Systems*, vol. 2, no. 3, August 1984.

[12] A. Rajimwale, V. Prabhakaran, and J.D. Davis, "Block Management in Solid State Devices," unpublished technical report, January 2009.

[13] A.S. Tanenbaum, J.N. Herder, and H. Bos, "File Size Distribution in UNIX Systems: Then and Now," *ACM SIGOPS Operating Systems Review*, vol. 40, no. 1 (January 2006), pp. 100–104.

[14] O. Troyanskaya, M. Cantor, G. Sherlock, P. Brown, T. Hastieevor, R. Tibshirani, D. Botstein, and R.B. Altman, "Missing Value Estimation Methods for DNA Microarrays," *Bioinformatics*, vol. 17, no. 6 (2001), pp. 520–525.

[15] S. Tweedie, "Ext3, Journaling Filesystem," *Ottowa Linux Symposium*, July 2000.

[16] D. Woodhouse, "JFFS: The Journalling Flash File System," *Ottowa Linux Symposium*, 2001.