IotaFS: Exploring File System Optimizations for SSDs

Henry Cook, Jonathan Ellithorpe, Laura Keys, Andrew Waterman
Computer Science Division, University of California at Berkeley
{*hcook, jde, laurak, waterman*}*@eecs.berkeley.edu*

Abstract

Solid-state drives (SSDs) are quickly becoming an affordable alternative to mechanical hard disk drives. SSDs perform low latency reads and have access times that are magnitudes smaller than those of their mechanical counterparts. Many current file systems make clever optimizations that target mechanical hard disk drives; the faster performance of SSDs without use of mechanical parts makes obsolete many of these file system optimizations. We created a file system meant for use with SSDs. Our file system IotaFS is based on the small, simple Minix file system, and we investigate a few optimizations of our own. We compare the performance of IotaFS with existing file systems such as ext3 and ReiserFS on both SSDs and mechanical hard drives.

1 Introduction

A solid-state drive (SSD) is a non-volatile storage device that uses flash memory rather than a magnetic disk to store data. SSDs are becoming increasingly cost efficient, and have begun to penetrate a variety of market segments once dominated by hard disk drives (HDDs). This trend has resulted in an increased interest in software and systems that utilize the characteristics of SSDs. We provide an analysis of the implications of SSDs for file system design by implementing several targeted optimizations in the context of a new, simple file system called **IotaFS**.

While SSDs provide some immediate performance benefits, many systems fail to realize their full potential due to assumptions made in software about the characteristics of the backing storage device. These assumptions might be about the speed at which the device will operate, or which code optimizations or management policies will yield improved performance. We find that modern popular file systems suffer suboptimal performance on SSDs due to all of the above, and we successfully mitigate some of the issues in IotaFS.

SSDs provide vastly improved read latency and random read/write access times relative to their mechanical hard disk counterparts. This is primarily due to the fact that SSDs do not need to seek across a physical, rotating disk in order to access their data. Most modern filesystems go to some lengths to reduce the time spent seeking across the disk by buffering up reads and writes, effort which unnecessarily increases latency for SSD drive accesses.

Unfortunately, SSDs must read and erase large (100KB+) blocks of data before performing any write to that block, leading to reduced write throughput and increased write latency in some cases. Write latency is still greatly reduced relative to that of HDDs. However, if forced to execute small writes to a diverse set of erase blocks, SSD performance will be reduced relative to writes to a spatially local area, due to the number of erase blocks which must be erased and rewritten in their entirety. Thus, the sequence of writes does not matter in the same way that it does for HDDs, but the distribution can have an impact on write performance.

A factor of concern to some system developers is that SSDs can support only a limited number of writes to individual disk blocks before the blocks 'wear out'. However, modern SSD drives include many extra blocks, and perform wear leveling across these internal to the device itself. Since the current generation of drives perform their own wear leveling, this duty does not have be performed by the file system, allowing for a simplification in function relative to past file systems targeting NAND flash memory devices.

The rest of this paper discusses our implementation of IotaFS, a simple filesystem that we use to explore the effect of various SSD-specific optimizations on file system performance.

2 Related Work

The increasing availability of flash memory has led to increased interest in designing file systems to take advantage of flash's fast access times. Sun's ZFS [5] was designed with the idea of directing frequent, small writes to a small flash-based log-intent device instead of to a larger storage pool. Oracle's btrfs [8] is a copy-on-write file system currently in development as a rival to ZFS; it has an optional SSD mode that will turn on as-of-yet undisclosed flash-specific optimizations. Two file systems meant specifically to run on flash devices are JFFS and YAFFS, though both were designed for use with raw flash devices and the MTD (Memory Technology Device) layer [2], not block devices like SSDs.

JFFS [4] is a log-structured file system designed for use with NOR flash. It handles wear-leveling for the underlying flash device through the use of a circular log. When the file system is mounted, its entire inode chain must be read into memory, making memory consumption proportional to the number of files in the system.

JFFS2 [1] is the successor to JFFS that added support for NAND flash and for compression, making it better suited for use with embedded devices. It also replaced the circular log wear-leveling scheme with a garbage collection algorithm and disallowed nodes to cross erase block boundaries.

YAFFS [3] was the first file system designed specifically for NAND flash. It is a journaling file system intended for use in embedded and handheld systems. It employs the use of "checkpointing" for fast mounting and has been ported to a number of operating systems.

The original Linux file system Minix was designed with the idea of keeping the source code clean and understandable; as such, it did not support any advanced features and only allows filenames up to 30 bytes in length and files up to 1 GB in size. Subsequent file systems contain more complex features that improve file system performance on mechanical hard drives; file systems such as ext2 organize blocks in a way that minimizes disk seeks.

The widely-used journaling file system ext3 uses hashed-tree directory indexes to decrease access time. However, it was created to be backward compatible with its predecessor ext2 and does not allow the use of block suballocation or extents, contiguous storage reserves meant to reduce fragmentation. A filesystem renowned for its efficiency in handling small files and directories containing large numbers of files is ReiserFS, which has a B+ tree as its underlying on-disk structure. It includes a number of advanced features such as tail-packing, dynamic inode allocation, and online resizing. The more advanced features a file system supports, however, the more unwieldy, and sometimes unstable, its code base becomes.

3 Basic Iota File System Design

We sought to implement a simple file system under the hypothesis that complex techniques used to improve performance on magnetic disks are not necessary for strong performance on SSDs. To that end, we based our design on the Minix v2 file system [9], which is fully-featured but not overly complex. In the sections that follow, we describe the logical organization of the file system and its layout on disk.

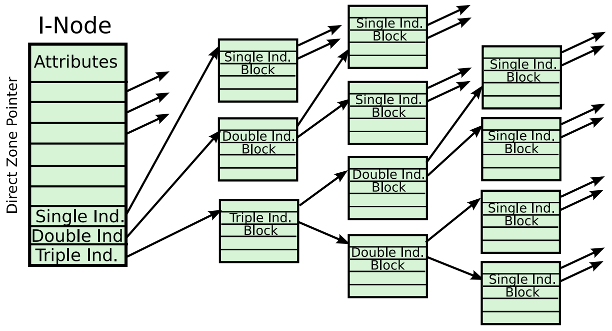

Figure 1: Iota's inode tree

3.1 File System Organization

An Iota file system resides on a single disk partition. It contains a single *superblock*, which contains the file system's vital statistics, such as the number of data blocks, and pointers to various on-disk structures. One of many *inodes* stores a file's parameters and contains pointers to its location on disk. Two *free lists* are bitmaps that store the locations of free inodes and data blocks. A *root block* points to a listing of the root directory's contents; all absolute path lookups begin here.

In addition to information like a file's size and permissions, an inode stores ten pointers. Seven of these point to *direct blocks*, which store the file's contents. There is a pointer to one *singly-indirect block*, which is a block that contains only pointers to direct blocks. There is also a pointer to one *doubly-indirect block*, which contains pointers to singly-indirect blocks. Finally, there is a pointer to one *triply-indirect block*, which contains pointers to doubly-indirect blocks. The resulting structure, depicted in figure 1, is logically a tree, whose leaves are file data blocks.

Assuming four-byte pointers (thus allowing for 2^{32} addressable data blocks) and a block size of B, a file can contain $7 + \frac{B}{4} + \frac{B^2}{16} + \frac{B^3}{64}$ blocks. Even with block sizes as small as 1KB, this organization allows for file sizes in excess of 16GB.

Directories are represented as files whose contents are one or more directory entries. A directory entry is a fixed-size structure that contains a file name and an inode number. Iota directory entries currently are 32 bytes and consist of a 2-byte inode pointer and a 30-byte filename. This structure, however, is parametrized, so a file system that consists of more than 2^{16} files or has longer file names is easily created.

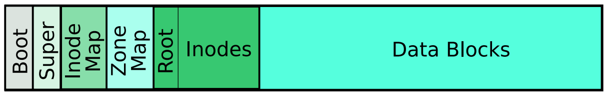

Figure 2: The on-disk layout of an Iota file system

3.2 On-Disk Layout

Iota's layout flattens the special blocks, free lists, and inode tree so that they exist in the disk's contiguous address space. It is represented in figure 2. The 0^{th} disk block is the boot block, and the 1^{st} is the superblock. Following them is the inode free list, which contains as many bits as there are inodes. The data block free list follows; it contains as many bits as there are data blocks, so the size varies with block size and disk size.

Following the data block free list are all inodes. They are statically allocated to simplify addressing. Each inode is 64 bytes; there are at most 2^{16} of them. All blocks following the inodes are data blocks.

4 Iota Optimizations Applied

Starting from the most basic implementation of a file system, we were able to find a particular area of improvement that allowed us to achieve performance on par with modern file systems including ReiserFS and Ext3. Essentially, we were able to take advantage of the high performance of random reads on an SSD to place written blocks of random files into convenient write locations to lessen the impact of erase block copying overhead, as well as the time spent looking for free blocks.

4.1 Erase Block Aware Allocation

SSDs are comprised of a hardware unit of allocation called *erase blocks*, which are often in excess of 128KB. When any portion of an erase block is written, the entire block must be re-written. So, unless enough contiguous data can be committed within a short temporal window, write bandwidth may suffer dramatically.

As a result, SSDs require spatial locality of writes for high throughput. While this is difficult to guarantee in general over a diverse set of workloads, heuristics may be applied to increase the likelihood that consecutive writes will target the same erase block. The technique we chose to this end is simple: in memory, keep track of the location of the last free block that was allocated. The next time a new block needs to be allocated, start scanning the free list from this location. Provided that free space in the file system is not heavily fragmented, consecutive writes (even to multiple files at the same time) are likely to target the same erase block.

A second benefit of this optimization is that it significantly decreases the compute time necessary to find a free block. If finding a free block is simply accomplished by scanning the free list from the beginning, it takes $O(n^2)$ time to write n blocks. While our approach has the same worst-case performance, we found it to be very effective in practice, so we ruled out using complex data structures to locate free blocks in asymptotically better time. Such techniques may also reduce the spatial locality of writes.

4.2 File Aware Allocation

As a further optimization to our erase block aware allocation scheme, we added an awareness of the files being written to in an attempt to co-locate disk blocks belonging to the same file. In the event of a file requiring a new block to be allocated, the file system finds the location of the last block allocated to that file in particular, and starts searching forward in the freelist from that position for the next free block. This scheme effectively increases the co-location of disk blocks for the same file in the event that many data blocks are written to in a file before moving on to the next file. Additionally, this technique preserves the improvement made by our previous allocation optimization of avoiding an $O(n^2)$ free list search.

In the event of high write concurrency, however, this method of disk block allocation significantly degrades write performance, increases the overhead of traversing kernel structures to find the last block written to for a file, and does a worse job of co-locating a file's data blocks (as high concurrency increases the probability of interleaving block allocations). Early performance measurements showed that this method in fact performed approximately as well as our stock file system implementation, due to the problems faced when multiple files are being written to simultaneously.

4.3 Increasing Disk Block Size

The standard disk block size in the linux kernel is 1KB, although file systems are allowed to change this if needed. For SSDs in particular, a larger block size can easily boost performance given that the drive must write to erase block sized chunks at a time no matter how few bytes must actually be written to that chunk. By writing larger blocks at a time, then, the file system naturally decreases the relative overhead of a single random write. A counter force to this optimization is of course wasted disk space for small files, which can be quite significant in many operating systems including UNIX, even for block sizes of 4KB [7].

4.4 Copy-On-Write

Copy-on-Write refers to a general memory updating technique that involves writing new data to a new location in memory rather that writing in place. Combined with write buffering and disk block remapping, this technique is particularly useful for flash drives as it allows the writer to minimize the overhead of the disk autonomously copying erase blocks on each write.

Although still in development, the copy-on-write technique that we are employing buffers page writes to the disk in memory until a full 128KB erase block size worth of data is ready to be written. At this point, block remapping code processes this buffer to aggregate these data blocks into a single erase block, finds a free erase block on disk, and pushes out the data to this location. The disk, in turn, will perform its own in-hardware wear leveling technique to find an appropriate free erase block for the data, reset the old erase block that was once used to store the data, and finish by remapping the data's logical address to the new physical location on the disk.

At first glance, there seem to be significant benefits of this technique. Considering random writes to the disk, this method of writing could potentially reduce 128 separate erase block remapping procedures performed by the disk into one, increasing write bandwidth due to significantly less overhead of erase block copying as well as increasing the lifetime of the disk. Furthermore, given that sequential reads and random reads to the disk are of nearly equal performance for flash drives, we expect that the performance degradation incurred due to the resulting data fragmentation will be negligible, although this remains to be tested.

Upon further analysis we find that the changing of data structures in the file system required during the data remapping procedure generates a non-trivial amount of overhead. In particular for each data block that is remapped, the indirect block that references the data block must be modified so that it points to the new block location. In the worst case, therefore, remapping 128 blocks would result in modification of 128 other disk blocks. Furthermore, the free list bit map structure kept by IotaFS to track locations of free disk blocks needs to be modified to reflect the migration.

For a 32GB flash drive, there are approximately four thousand 1KB disk blocks required to maintain this free list, and so it is possible that 128 separate blocks could be modified via the clearing of old location bits, as well as 1 additional block for setting 128 contiguous bits representing the new erase block-sized location. Cumulatively, then, 258 total erase blocks may be touched during the effort to coalesce 128 1KB blocks into a single erase block. We predict that in a typical scenario, however, it is unlikely that such a spread of accesses will be realized due to spacial locality of data accesses, and this spacial locality translating into making modifications to the same indirect blocks.

Our in-progress implementation of this copy-on-write technique manifests itself in the pagewrite() method of the file system's address space object to intercept page writes to the disk, buffer an erase block worth of data, and then modify the necessary data structures before pushing the buffer to an erase block. All modified file system structures are dirtied in the process, whereupon the kernel takes care of pushing these out to disk appropriately.

5 Evaluation

We ran both a synthetic and a realistic set of benchmarks on a number of file systems to compare their performance on SSDs. The same set of experiments was then run on a hard disk drive to see how performance is affected by mechanical-drive-based file system optimizations. The file systems under scrutiny were ext2, ext3, ReiserFS, Minix, Iota, Iota with 4KB block size, and Iota with 4k block size and the fast allocation optimization.

The tests were conducted on a 2 Quad-Core AMD Opteron 8356 machine running Linux 2.6.18 at 2.3 GHz with 8 GB of memory and an Intel X25-E SATA solid state drive containing 32 GB of SLC NAND flash memory. The SSD has a read bandwidth of 250 MB/s and a write bandwidth of 170 MB/s and has access times of 75 us. The hard drive used is a 147 GB Fujitsu MBB2147 SAS with a data transfer rate of 3 Gb/s. It has an average seek time of 4 ms for reads and 4.5 ms for writes and an average latency of 2.99 ms.

5.1 Synthetic Benchmark: Bonnie++

The synthetic benchmark suite Bonnie++ was used for the first set of tests. Bonnie++ measures file system performance for sequential reads, writes, re-writes, and random seeks across 1 GB files and then measures creating, reading, and deleting a large number of small files both sequentially and randomly. We ran Bonnie++ on 16 1 GB files in fast mode, skipping reading and writing by single characters and instead reading and writing by blocks. The realistic workload used for the second set of tests was creating a tarball of the Linux kernel source code, which consists of a large number of small reads followed by a large sequential write, and then untar-ing the code.

Results of the synthetic benchmark are shown in figure 3. The sequential write task in Bonnie++ revealed a huge performance gap between the optimized file systems and the simple ones, with file systems like ext3 and ReiserFS having a throughput 30x that of Minix and Iota. However, Iota with 4KB block size and a fast allocation policy narrows the gap significantly, obtaining about 70% of ReiserFS's throughput. This surprising result shows how inefficient the current Minix allocation policy is.

The re-write and sequential read tasks show the benefits of larger block size in Iota. The highly optimized file systems ext2, ext3, and ReiserFS all nearly saturate the SSD's read bandwidth, due in part to their on-disk

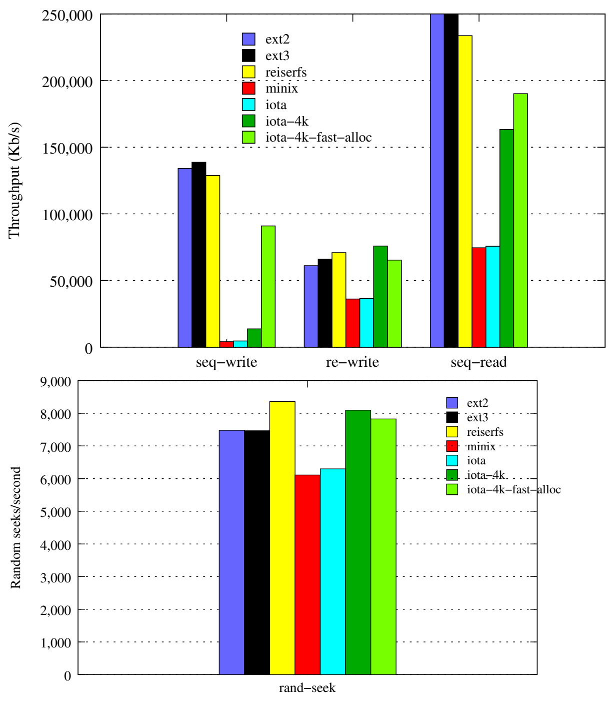

Figure 3: Bonnie++ Benchmark Results

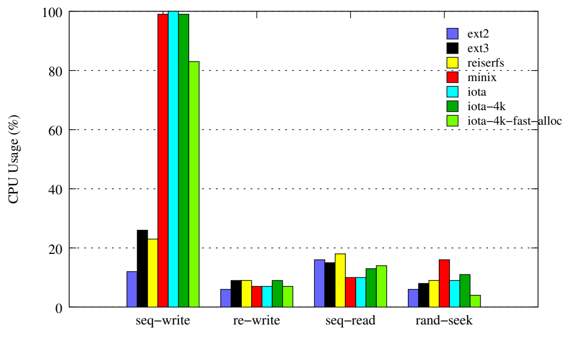

Figure 4: CPU Usage in Bonnie++ Benchmark

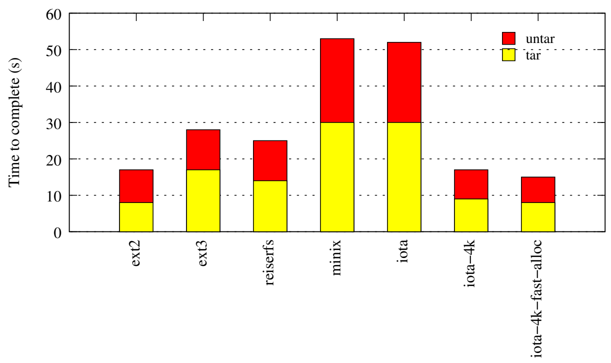

Figure 5: Realistic Benchmark Results

structures. Not surprisingly, Minix and Iota only achieve about 30% of the read bandwidth, as they do not use an optimized traversal of the underlying structure. However, the two versions of Iota with 4KB block size improve that measurement to about 70% of the read bandwidth simply by virtue of transferring more data with each I/O call and reducing the number of indirect blocks accessed.

All seven file systems were able to perform a comparable number of random seeks within files. The fast access and seek times provided by an SSD allow the Minix and Iota file systems to perform just as well as the file systems that are highly optimized to minimize random read times on mechanical drives. As we demonstrate in a later experiment, this test has very different results when run on a hard disk drive instead of an SSD.

As seen in figure 4, the CPU usage results of the sequential write test show a definite need for improvement in Minix and Iota. Minix, Iota, and Iota with 4KB block size all saturate the CPU in their quest to locate free inodes. The more efficient, improved allocation policy for Iota reduces the CPU usage by nearly 20%, but even its reduced usage is 4 to 8 times that of ext2, ext3, and ReiserFS.

5.2 Realistic Workload Benchmark: Tar and Untar

The realistic benchmark of tar-ing and then untar-ing the Linux kernel source code proved to be a test in which Iota with optimizations actually outperforms ext3 and ReiserFS. The results of these two stages are shown in figure 5. Optimized Iota performs nearly four times as well as Minix and unoptimized Iota. The two variants of Iota with optimizations perform as well as ext2 and nearly twice as well as ext3 and ReiserFS. This benchmark demonstrates how much journaling, and hence doubling the number of writes required, can impact file system performance. The performance of optimized Iota being comparable to ext2, ext3, and ReiserFS is even more impressive when considering the lines of C code used to implement each file system, Iota having 1/2 the code of ext2, 1/3 the code of ext3, and 1/10 the code of ReiserFS, as can be seen in figure 6.

5.3 Bonnie++ on a HDD

To test how important SSDs are for the performance of our Iota optimizations, we ran Bonnie++ on a hard disk drive. We compared the file systems ext2, ext3, Reis-

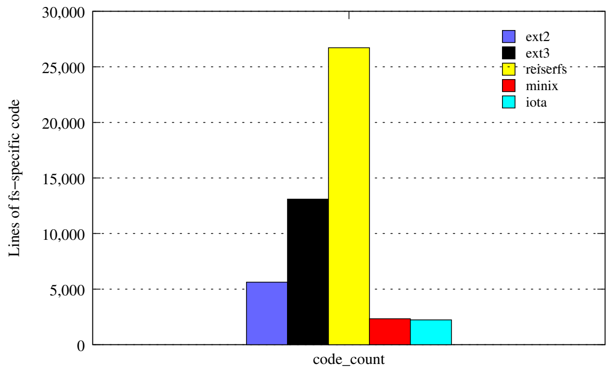

Figure 6: Lines of Source Code

erFS, and Iota with 4KB blocks and fast block allocation, the results of which are shown in figure 7.

The results of this test are not surprising. The non-journaling ext2 outperforms its journaling counterparts ext3 and ReiserFS on writes, and the three perform similarly for re-writing, sequential reading, and random seeks. Optimized Iota's performance suffers drastically from being on a mechanical drive. Particularly notable is its poor performance for sequential reads, a task for which the other three file systems were carefully designed to increase performance. Another test in which having Iota on an SSD makes a difference is random seeking; on the SSD, Iota attained the same throughput for random seeks as the other three file systems thanks to flash, but when those random seeks happen on a HDD, Iota's complete lack of consideration for a mechanical drive curtails its performance significantly. The difference in throughput between the HDD-optimized file systems and Iota is made very clear from this benchmark.

After seeing these results, it's easy to understand why file system creators in the past have abandoned code simplicity for performance. Having only mechanical disk drives to base their performance considerations on, they put careful effort into features and optimizations that new SSDs with their fast access times render unnecessary. The stark difference in how well Iota compares to other file systems on SSDs and HDDs suggests that new file systems can be made simpler and with less source code while still enjoying increased performance due to the low latency of flash memory.

6 Optimizations Not Applied

In considering various changes to the file system to improve its performance, we found that these changes either would not be well suited for solid state disks, or simply did not make sense for our file system. Additionally, given that the cpu time spent in the file system itself became a significant factor for performance when

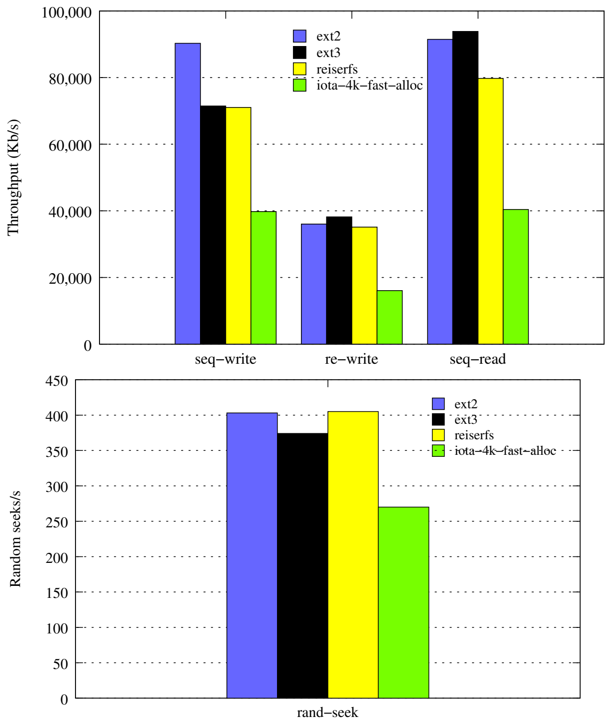

Figure 7: Bonnie++ Results on a HDD

using SSDs, we felt that keeping the critical code path short was an optimization in itself and thus precluded improvements that were unlikely to compensate for the additional time spent in the kernel.

6.1 Prefetching

Workloads with little concurrency suffer from poor performance because they often cannot tolerate the latency of I/O operations. A common remedy is to prefetch data to hide such long-latency events. On an SSD, however, there is much less latency to hide: access time is reduced by more than an order of magnitude over a magnetic disk. Further, Little's Law[6] dictates that the amount of concurrency we need to achieve the same throughput is proportional to the access time of the device. So, many workloads that may have benefited from prefetching on a magnetic disk will not see gains from prefetching on an SSD.

Prefetching complicates the implementation of the file system for what is not likely to be a significant performance improvement, so we chose not to implement it.

6.2 Alternative Indirect Block Structure

With respect to structures for locating data blocks, one option that we considered was to use a linked list of indirect blocks instead of the typical tree structure. This has the advantages of being simple, appropriate for sequential reads, allowing for unbounded file sizes, and having low overhead. The low overhead, in particular, results from being able to map 255 data blocks per indirect block in the linked list (the 256th entry is used as the pointer to the next indirect block). Thus, we achieve an average data to meta data ratio of 255:1. Using a more complicated scheme using triply indirect blocks achieves nearly the same efficiency for large files. Ultimately, we did not implement the linked list scheme as the loss in performance when reading/writing randomly to a large file was simply too large. A back of the envelope calculation showed that to perform a cold-start read of the last byte of a 1GB file would take approximately 320ms given the parameters of our SSD. Thus, we decided that it would be better to use a more complicated multi-level lookup scheme to allow for approximately equal meta data overhead for large files and much better random read and write performance.

6.3 Using a Write-Through Cache

Given the low latency of the disk, we considered using a write-through buffer cache that would have written every block to the disk upon being marked as dirty. The reason for doing this would simply be to avoid the overhead caused by having to store dirty entries in the buffer and keep track of them. For large data writes, however, this technique would cause severe erase block overhead since we are unable to stream sequential block writes to the disk.

7 Future Work

There are several paths that future work in this area might take. We can do more experiments to test the effectiveness of more complicated file system designs. We can also work to improve the portability and manageability of the file system to allow for the possibility of wider adoption and open source development.

The biggest improvement that we were unable to complete by the end of this semester was to finish adding copy-on-write functionality to IotaFS. Copy-on-write could have significant performance benefits for SSDs, since if managed correctly it could result in fewer erases and rewrites being required. However, applying this technique effectively on SSDs requires buffering up writes, which as we saw has negative effects on write bandwidth utilization.

Further experiments could also include finding the ideal file system block size for different types of drives and workloads. Our analysis in this area suggests that block size can have a large impact on performance, though of course increasing block size is likely to have a reduced rate of returns. Further effort spent on improving the block allocation algorithm could also produce significant gains, especially considering the CPU load apparently generated by some write benchmarks. We would also like to measure system performance over a long time frame, to measure durability, and whether the effects of file fragmentation are mitigated by SSDs as we would expect.

A set of experiments along a different path would be to consider other use cases for the SSDs. For example, researchers in the UC Berkeley RADLab have expressed interest in studying the performance isolation implications that SSDs have for disk I/O on machines with multiple resident VMs. On such systems with HDD-based storage, it is difficult to isolate I/O performance due to the massive increase in the entropy of access locality that occurs when the VMs are working on different data sets. SSDs should have no such limitation, but this might not be adequately demonstrated using an unoptimized file system. IotaFS provides a mechanism to conduct experiments of this type.

In order to make our work accessible to others, we will improve the manageability and compatibility of IotaFS. One thing we will have to provide is a `fsck` utility for recovering data from disks which, due to lost power, have been left in potentially inconsistent states. We will also need to improve the portability of the code to other architectures and improve the compatibility of the module to match current and future kernel versions. For the most part, none of these change should require significant code alterations to the baseline system.

Overall, our early optimizations have already shown significant improvements over the baseline design and performance that is comparable to file systems already in wide usage. Further work should allow us to increase the performance gap and gain additional insight into the optimizations required for ideal SSD performance.

8 Conclusion

Solid state drives have a number of interesting characteristics that change the access patterns required to optimize metrics such as disk lifetime and read/write throughput. In particular, SSDs have approximately two orders of magnitude improvement in read and write latencies, as well as a significant increase in overall bandwidth. Our hypothesis in beginning this research was simply that the complex optimizations applied in current file system technology doesn't carry over to SSDs given such dramatic changes in performance characteristics. To explore this hypothesis, we created a very simple file system research vehicle, IotaFS, based on the incredibly simple and small Minix file system, and found that with a few modifications we were able to achieve comparable performance to modern file systems including Ext3 and ReiserFS, without being nearly as complex.

The optimizations explored using IotaFS included using an erase block aware data allocation scheme, experimenting with a file-aware allocation method, increasing the disk block size, and implementing a copy-on-write data buffering technique. We found that allocating new data blocks sequentially during a write was the most significant performance improvement made to our file system, which effectively reduced erase block copying overhead required during each write to the disk. Additionally, although the data blocks for each file become fragmented on disk as a result of this technique, the SSD's ability to read just as well randomly as it can sequentially significantly reduces the impact of this fragmentation. Furthermore, increasing the disk block size from 1KB to 4KB had a positive impact on performance by furthering the reduction of erase block overhead during a write when writing randomly. Ultimately, even with our resulting simple file system, we were able to achieve performance comparable to that of a highly optimized and complicated modern file system.

There were also a few optimizations that we did not explore including using prefetching, changing the indirect block structure, and using a write-through buffer cache. Prefetching was determined to be less useful for SSDs as the access times to the disk are so small that read and write concurrency would not have to be nearly as high as required to reach near peak bandwidths on mechanical hard drives. Using a write-through cache, on the other hand, causes high response times and thus low performance for single file access, as well as low overall performance in the event of low concurrency.

While this remains to be a rich area of research, there are a few avenues that we feel are particularly worth exploring in the future. Finishing the copy-on-write technique would allow us to explore re-allocating disk blocks in an erase-block aware fashion. This has the benefit of maximizing write bandwidth to the disk, but as discussed in the paper, however, this technique can also potentially lead to more than a doubling of disk writes. On the other hand, many writes to the same file reduces this overhead while effectively coalescing randomly dispersed data blocks. A characterization of our rudimentary implementation would be required to determine the necessary optimizations to reduce its overhead. Of course, additional improvements to our block allocation policy as well as finding the right block size would likely lead to further increases in performance.

In exploring the use an extremely simple file system for flash drives, and finding that we were able to achieve comparable performance to modern file systems by making only a few modifications particular to the details of solid state disks, we feel that exploring this field further will lead to even greater performance over other heavily optimized file systems available today.

9 Acknowledgements

Research supported by Microsoft and Intel funding (Award #20080469). Many thanks to Jon Kuroda for his numerous trips into the Black Box on our behalf and for being generally helpful.

References

[1] *JFFS2: The Journalling Flash File System, version 2.*

[2] *Memory Technology Device (MTD) Subsystem for Linux.*

[3] *YAFFS: Yet Another Flash File System.*

[4] A. Communications. *JFFS: Journalling Flash File System.*

[5] A. Leventhal. *"Flash Storage Memory", Communications of the ACM, July 2008.*

[6] J. D. C. Little. A proof of the queueing formula $l = \lambda w$. *Operations Research, 9, 383-387*, 1961.

[7] M. K. Mckusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A fast file system for unix. *ACM Transactions on Computer Systems*, 1984.

[8] Oracle. *BTRFS.*

[9] A. S. Tanenbaum and A. S. Woodhull. *Operating Systems Design and Implementation (3rd Edition) (Prentice Hall Software Series).* Prentice Hall, January 2006.