Stupid File Systems Are Better

Lex Stein
Harvard University

Abstract

File systems were originally designed for hosts with only one disk. Over the past 20 years, a number of increasingly complicated changes have sought to optimize the performance of file systems on a single disk. At the same time, separated by a narrow interface known as the block, storage systems have been advancing on their own. Storage systems increasingly use parallelism to provide higher throughput and fault-tolerance and employ additional levels of address mapping indirection (known as virtualization) to improve system flexibility and ease administration costs. In this paper, I show that file and storage systems have gone in different directions. If file systems want to take advantage of modern systems such as disk arrays, they could do better by throwing out the past 20 years of optimizations and simply layout blocks randomly.

1 File Systems

The first popular file systems were on the local disks of host computers. Today there are often several hops of networking between a host and its permanent storage today, but most often, the final destination or original source of data is a hard disk. Disk geometry has played a central role in the past 20 years of file system optimizations. The first file system to make allocation decisions based on disk geometry was the BSD Fast File System (FFS) [5]. FFS improved file system throughput over the UNIX file system by clustering sequentially accessed data, colocating file inodes with their data, and increasing the block size, while providing a smaller block size, called a fragment, for small files. FFS introduced the concept of the cylinder group, a three-dimensional structure consisting of consecutive disk cylinders, and the foundation for managing locality to improve performance. After FFS several other advances were made improving file system layout to bring the on-disk locality closer to the access pattern.

Log-structured file systems [7] [8] take a fundamentally different approach to data modification that is more like databases than traditional file systems. An LFS updates copy-on-write rather than update-in-place. While an LFS, looks very different, it is motivated by the same assumption as the FFS optimizations; the high performance of sequential operations. Advocates of LFS argued that reads would become insignificant with large buffer caches. Using copy-on-write necessitates a garbage collection thread known as the cleaner.

Journaling is less radical than log-structuring and is predicated on the same assumption of efficient sequential disk operations. With a log-structured file system, a single log stores all data and metadata. Journaling stores only metadata intent records in the log and seeks to improve performance by transforming metadata update commits into sequential intent writes, allowing the actual in-place update to be delayed. The on-disk data structures are not changed and there is no cleaner thread. Soft updates [2] is a different approach that aims to solve the same problem. Soft updates adds complexity to the buffer cache code so that it can carefully delay and order metatadata operations.

All of these file system advances have been predicated on the efficiency of sequential disk operations. Does this hold for current storage systems?

2 Storage Systems

File systems use a simple, narrow, and stable abstract interface to storage. While the exact driver used to implement this interface has changed from IDE to SCSI to Fibre Channel and others, file systems have continued to use the put and get block interface abstraction. File and storage system innovation has progressed on the two sides of this narrow interface. While file systems have developed more and more optimizations for the single disk model of storage, storage systems have evolved on

their own, and have changed substantially from that simple single disk.

The first big change was disk arrays and, in particular, arrays known as Redundant Arrays of Inexpensive Disks (RAID). A paper by Patterson and Gibson [6] popularized RAID and the beginnings of an imperfect but useful taxonomy called RAID levels. RAID level 0 is simply the parallel use of disks with no redundancy and is often called Just a Bunch Of Disks (JBOD). JBOD and RAID systems employ disks in parallel to increase system throughput. Disk arrays typically stripe the block address space across their component disks. For large stripes, blocks that are together in numerical block address space will most likely be located on the same disk. However, a file system that locates blocks that are accessed together on the same disk will be prohibited from operating on blocks in parallel. For a system that translates temporal locality to numerical block address space proximity there are two opposing forces struggling here; (1) an increasing stripe size will cluster blocks together and improve single disk performance, and (2) the increasing stripe size will move blocks that are accessed together onto the same storage device, eliminating the opportunity for parallelism.

Storage systems are expensive to manage. Reducing management costs is an important goal for the design of storage systems, just like improving file system metadata throughput is a goal for the design of file systems. This has led to a phenomenon known as storage virtualization. In a primitive form, virtualization is present in SCSI drives, where disk firmware remaps faulty blocks. However, this remapping is not believed to be significant enough to meaningfully disrupt the foundational assumptions of local file system performance. On the large scale, virtualization is used to provide at least one layer of address translation between the file system and the storage system. This indirection gives storage administrators the freedom to install new storage subsystems, expand capacity, or reallocate partitions without affecting file system service. This is great for storage system and administration, but it completely disrupts the assumption that there is a strong link between proximity in the block address space and lower sequential access times through efficient mechanical motion.

3 Experimental Approach

This paper is motivated by the question: how do the optimizations of local file systems affect their performance on advanced parallel storage systems? To answer this, I have traced several workloads and run these traces through a storage system simulator. The linux Ext2 and Ext3 file systems are both advanced current file systems that incorporate many locality optimizations. Ext3 journals and, in my experiments, Ext2 does not. To find out how a file system would perform that does not carefully and cleverly allocate blocks within the block address space, I have generated traces that are random permutations of the Ext2 and Ext3 block traces. Block identities are preserved under permutation. Only the block address changes. The degree of the permutation is known as the *stupidity* of the trace. Within a generated trace, each block is permuted (no replacement) with constant probability p to a new address in the block address space. The new addresses are selected with uniform probability from those that remain free. If p is 1.0, then every block is permuted and the trace is known as the *stupid* trace. A stupid trace represents the workload on a file system that does not (or cannot) put any thought into block layout. If p is 0.0, then no block is permuted and the trace is exactly the same as the original Ext2 or Ext3 trace. Traces with p of 0.0 are known as *smartypants* traces because they represent the workload of a file system that spends a lot of effort being clever about block layout.

All the trace generation experiments were run on the same Linux 2.4.17 system. Throughout the tracing, the configuration of the system was unchanged, consisting of 32K L1 cache, 256K L2 cache, a single 1GHz Intel Pentium III processor, 768MB DRAM, and 3 8GB SCSI disks. The disks are all Seagate ST318405LW and share the same bus. One of the disks was dedicated to benchmarking, storing no other data and used by no other processes.

The lltrace [4] low-level disk trace patch for Linux traces block operations just before they are sent to disk. Lltrace records the sector number, the number of sectors requested, and whether the operation is a read or a write. Lltrace traces to an in-kernel buffer. I used lltrace to trace the block operations of my benchmarks. Once the workloads had completed, I would dump the trace buffer into a file on an untraced device.

Two benchmarks were run to generate the block traces; postmark and linux-build.

Postmark [3] is a synthetic benchmark designed to use the file system as an email server does, generating many metadata operations on small files. Postmark was run with file sizes distributed uniformly betwen 512B and 16K, reads and writes of 512B, 2000 transactions, and 20000 subdirectories. The postmark trace was generated from a run with these parameters on an Ext2 file system.

Linux-build is a build of a Linux version 2.4.17 preconfigured kernel. It is a real workload, not a synthetic benchmark. Linux-build was used to generate two block traces. There was only one configuration difference between the two runs of linux-build. One was run on Ext2 and the other on Ext3. I did this to investigate the impact of journaling.

All experimental approaches to evaluating computer

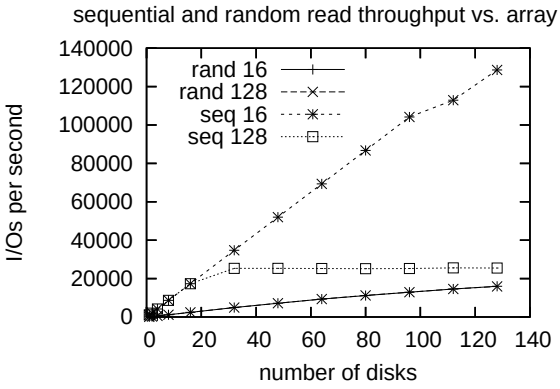

Figure 1: read microbenchmarks: throughput of sequential and random reads for stripe units of 16 and 128 sectors (8KB and 64KB).

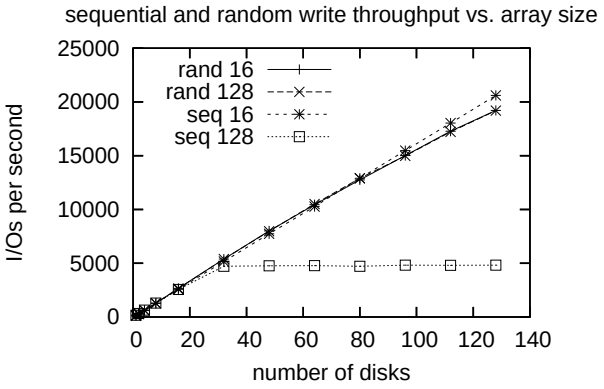

Figure 2: write macrobenchmarks: throughput of sequential and random writes for stripe units of 16 and 128 sectors (8KB and 64KB).

systems have their strengths and weaknesses. Trace-driven simulation is one kind of trace-driven evaluation. The central weakness of trace-driven evaluation is that the workload dows not vary depending on the performance of the system. On the other hand, it's central strength is that it represents something meaningful. A recent report by Zhu et al. discusses these issues [9].

In order to evaluate the performace of these workloads on a variety of modern storage systems, I built a simulator. The simulator simulates the performance of RAID arrays, and uses a disk simulator, CMU DiskSim [1], as a component to simulate disks. CMU DiskSim is distributed with several experimentally validated disk models. Every experimental results reported in this paper was generated using the validated 16GB Cheetah 9LP SCSI disk model[1].

4 Experimental Results

4.1 Microbenchmarks

I ran several microbenchmarks. The behavior of these microbenchmarks help to explain the behavior of the macrobenchmarks. All the benchmarks issue I/O to the array controller with a window of 200 asynchronous I/Os. The sequential read and write benchmarks issue I/Os sequentially across the block address space exported by the array controller. The random read and write benchmarks issue I/Os randomly across the block address space. In every benchmark, I/Os are done to the array controller in independent 4KB chunks. The microbenchmarks were run for different stripe units.

Sequential far outperfoms random for the smaller stripe units. This is due to track caching on the disk. Sequential rotates across the disks. When it recycles, the blocks are already waiting in the cache. For the smaller stripe units, the track cache will be filled for more cycles. Larger stripe units benefit less from spreading the parallel I/Os across more disks. As the stripe unit becomes greater, the benefit of the track caching becomes less and less of a component, bringing the sequential throughput down to the random throughput.

Writes do not benefit from track caching. Without track caching, sequential and random have similar performance for small stripe units across disk array sizes. As the stripe unit increases, sequential I/Os concentrate on a smaller and smaller set of disks. Random performance is resilient to the stripe unit. This shows how performance is stable across different levels of virtualization when spatial locality is removed from the I/O stream. This effect is responsible for the behavior we will see next in the macrobenchmark section.

4.2 Macrobenchmarks

In this section, I will discuss the results of 3 experiments.

The first experiment measured the run time of the stupid and smartypants Ext2 postmark traces on JBODs of size varying between 1 (not really a bunch of disks) and 256 disks. This experiment investigates how the relative scaling of the traces with array size. A JBOD is an array of disks with no redundancy. There is no parity and therefore no parity computation. In this experiment, the trace block address space was striped evenly across all the disks in the array. Each disk had only one stripe unit and that stripe unit was of size equal to the full space divided by the number of disks. Figure 3 shows a pattern observed across similar experiments with the other workloads. For 1 or a small number of disks, smartypants dominates performance. For example, with 1 disk, stupid takes 48% longer to run the postmark benchmark. However, with 2 disks, stupid is already faster, taking 20%

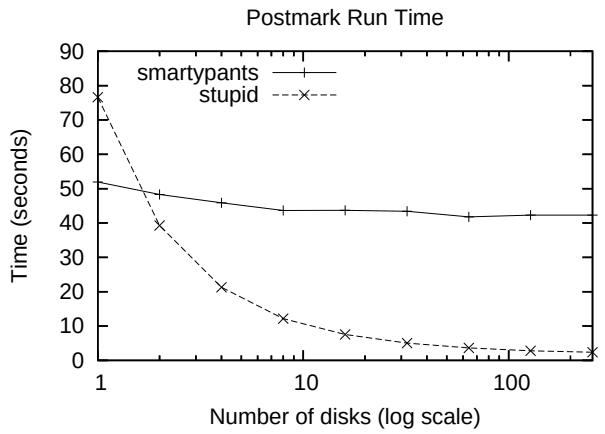

Figure 3: Postmark ext2 run time across JBOD (Just a Bunch Of Disks) arrays of varying size.

less time than smartypants. The performance of smartypants is quite stable across increasing disk sizes. The gap between smartypants and stupid grows quickly on this benchmark because of the high degree of spatial locality of the postmark benchmark operations. The random permutation of the stupid trace spreads the blocks evenly accross the disks allowing for parallelism. The locality clustering of smartypants leads to convoying from disk to disk as smartypants jumps from one cluster to another, bottlenecked on the internal performance of the single disks.

Figure 4 shows the results of the second experiment. This experiment simulated the Ext2 and Ext3 stupid and smartypants traces on RAID-4 and RAID-5 arrays. In all cases, smartypants loses to stupid. Recall that Ext3 journals while Ext2 does not. The largest improvements for stupid is on both the Ext2 and Ext3 RAID-5 systems. In both cases, RAID-5 smartypants underperforms RAID-4 smartypants while RAID-5 stupid outperforms RAID-4 stupid.

Figure 5 shows the normalized performance of a single disk and a RAID-5 array for varying stupidity of the Ext3 linux build. Figure 5 shows, among other things, that smartypants really does help with single disk systems. Moving from stupid (stupidity of 1) to smartypants (stupidity of 0) the performance on the single disk improves, with the run time decreasing by 59%. The RAID-5 pattern is completely different. As stupidity decreases, the run time increases by 35%. This result shows the different effect of block allocation choice on parallel and centralized devices. A parallel storage device can benefit from eliminating the locality-based block allocation that improves single-disk file systems.

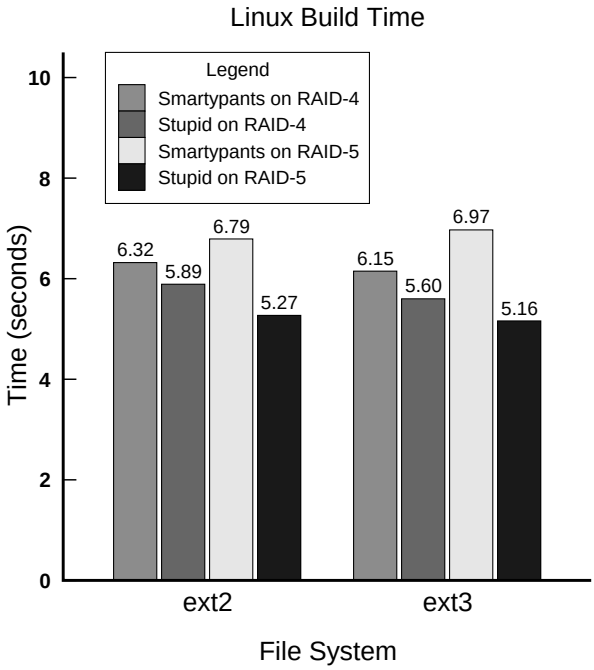

Figure 4: Build time of ext2 and ext3 smartypants and stupid traces on RAID-4 and RAID-5 arrays. Both arrays have stripe unit of 128KB and 256 disks. For both, the group size is 256.

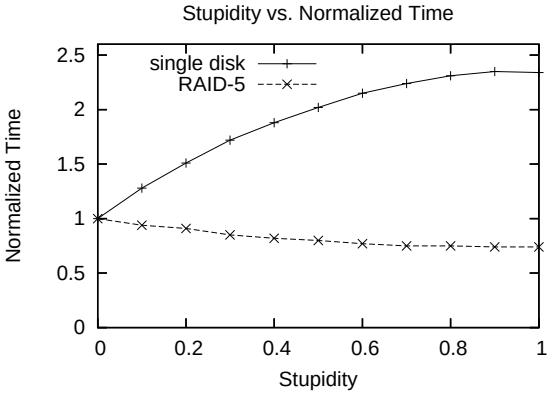

Figure 5: Stupidity vs. normalized time for the ext3 linux-build workload on a single disk and a 256 disk RAID-5 array.

5 Conclusions

The experiments described in this paper have shown that random block allocations will often perform better on a storage array than the careful block allocation decisions of file systems such as Ext2 and Ext3. The randomized permutations scramble all the careful block proximity decisions of these advanced file systems, yet outperform them on disk arrays under several different workloads. It's time we rethink file systems.

6 Acknowledgments

Discussion with Margo Seltzer originated the question motivating this paper. Discussion with Daniel Ellard contributed insight on the workings of disk arrays.

References

[1] BUCY, J. S., GANGER, G. R., AND CONTRIBUTORS. The disksim simulation environment version 3.0 reference manual. Tech. Rep. CMU-CS-03-102, CMU, January 2003.

[2] GANGER, G. R., AND PATT, Y. N. Metadata update performance in file systems. In *Proceedings of the USENIX 1994 Symposium on Operating Systems Design and Implementation* (Monterey, CA, USA, 14–17 1994), pp. 49–60.

[3] KATCHER, J. Postmark: A new filesystem benchmark. Tech. Rep. TR3022, Network Appliance, 1997.

[4] LLOYD, O. Lltrace: Low level disk trace patch for linux, January 2002.

[5] MCKUSICK, M. K., JOY, W. N., LEFFLER, S. J., AND FABRY, R. S. A fast file system for UNIX. *Computer Systems 2*, 3 (1984), 181–197.

[6] PATTERSON, D., AND GIBSON, G. A case for redundant arrays of inexpensive disks (raid). In *SIGMOD International Conference on Management of Data* (June 1998).

[7] ROSENBLUM, M., AND OUSTERHOUT, J. K. The design and implementation of a log-structured file system. *ACM transactions on computer systems 10*, 1 (Feb. 1992), 26–52.

[8] SELTZER, M. I., BOSTIC, K., MCKUSICK, M. K., AND STAELIN, C. An implementation of a log-structured file system for UNIX. In *USENIX Winter* (1993), pp. 307–326.

[9] ZHU, N., CHEN, J., CKER CHIUEH, T., AND ELLARD, D. Scalable and accurate trace replay for file server evaluation. Tech. Rep. TR153, SUNY Stony Brook, December 2004.

Notes

[1] This is a circa-1999 disk. I am completing the validation and integration of a more recent disk.