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 optimized the performance of file systems on a single disk. Over the same time, storage systems have advanced on their own, separated from file systems by the narrow block interface. Storage systems have increasingly employed parallelism and virtualization. Parallelism seeks to increase throughput and strengthen fault-tolerance. Virtualization employs additional levels of data addressing indirection to improve system flexibility and lower administration costs. Do the optimizations of file systems make sense for current storage systems? In this paper, I show that the performance of a current advanced local file system is sensitive to the virtualization parameters of its storage system. Sometimes random block layout outperforms smart file system layout. In addition, random block layout stabilizes performance across several virtualization parameters. This approach has the advantage of immunizing file systems to changes in their underlying storage systems.

1 File Systems

The first popular file systems used local hard disks for persistent storage. Today there are often several hops of networking between a host and its persistent storage. Most often, that final destination is still a hard disk. Disk geometry has played a central role in the past 20 years of file system development. 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 earlier 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 basis for managing locality to improve performance. After FFS, several other advances further optimized file system layout and access for single disks.

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, its design is motivated by the same assumption as the FFS optimizations. That is, sequential operations have the best performance. Advocates of LFS argued that reads would become insignificant with large buffer caches. Using copy-on-write necessitates a cleaner thread to read and compact log segments. The behavior of log-structured file systems is still incompletely understood and the subject of ongoing research.

Journaling [3] is less radical than log-structuring and is predicated on the same assumption that sequential disk operations are the most efficient. In 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.

These advances have been predicated on the efficiency of sequential operations in a block address space. Does this hold for current storage systems?

2 Storage Systems

File systems use a simple, narrow, and stable abstract interface to storage. While the underlying system implementing 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 have progressed independently 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 evolved substantially from that 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 outlined the beginnings of an imperfect but useful taxonomy called RAID levels. RAID level 0 is simply the parallel use of disks with no redundancy. Arrays employ disks in parallel to increase system throughput. They typically stripe the block address space across their component disks. For large stripes, blocks that are together in the numerical block address space will most likely be located together on the same disk. However, a file system that locates blocks that are accessed together on the same disk will prohibit the storage system from physically operating on those blocks in parallel. For a file system that translates temporal locality to numerical block address space proximity two opposing forces are in struggle. First, an increasing stripe unit will cluster blocks together and improve single disk performance. Second, an increasing stripe unit will move blocks that are accessed together onto the same storage device, reducing the opportunity for mechanical parallelism.

Storage virtualization is just a level of indirection. A translation layer does not come for free. Why are storage systems becoming increasingly virtualized? What problem is this solving? Virtualization abstracts the block address space to hide failures and facilitate transparent reconfiguration. By hiding failures, the system can use more components to achieve higher throughput, as with arrays. By allowing for transparent reconfiguration, the system can both reduce administration costs and increase reliability by allowing administrators to upgrade systems without notifying applications. Administrators can install new storage subsystems, expand capacity, or reallocate partitions without affecting file system service. The indirection of virtualization is great for storage system scalability 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.

From the outside, a storage system's virtualization looks like one monolithic map. On closer inspection, it is a layering of mappings that compose to take an address from the file system down to where it actually represents a location in physical reality. At each translation level, logical addresses are exported up and physical addresses are sent down.

```
disk:
 paddr = tbl[baddr / chunk_sz]
           + baddr % chunk_sz
array:
 (diskno, paddr) =
   fun(stripe_unit, numdisks, baddr)
volume:
 (diskno, paddr) =
   fun(stripe_unit, numdisks,
     tbl[baddr / chunk_sz]
       + baddr % chunk_sz)
```

Figure 1: **The virtualization models**
This figure shows pseudocode for the disk, array, and volume virtualization models. These models map the block addresses used by the file system to the physical addresses used internally by the storage system.

Virtualization is present in SCSI drives, where firmware remaps faulty blocks. However, at least in young and middle-aged drives, this remapping is not believed to be significant enough to meaningfully disrupt the assumptions of local file systems. One set of experiments in this paper investigates how a particular model of disk remapping affects performance. On a scale larger than single disks, virtualization is used to provide at least one level of address translation between the file and storage systems. Arrays remap addresses for striping and volumes further remap partitions across devices.

Figure 1 shows the model of virtualization used in this paper. The `baddr` is the block address used by the file system and the `paddr` and `diskno` are, respectively, the physical sector address and disk number used within the storage system. The virtualization layer maps file system block addresses to storage system physical sector addresses. The `chunk_sz` is the size of the virtualization chunk, in sectors. Chunks are remapped between virtual and physical address spaces maintaining the ordering of their internal sectors. Likewise, the `stripe_unit` is the size of the stripe and stripes are remapped maintaining their internal sector ordering. Chunking of volumes requires memory to store the individual mappings. Here this is represented as a table, `tbl`. The table is indexed into using integer division on block addresses to number chunks from base 0. The physical addresses in the table represent the base of the chunk. The block address modulo the chunk size is added to this base for the physical sector address. The deterministic remapping of striping can be computed with a function, shown here as `fun`. This function takes the stripe unit, the size of the array, `numdisks`, and the block address and outputs the sector and disk index.

3 Experimental Methodology

This paper is motivated by the question: how do file systems optimizations affect their performance on virtualized storage systems?

To answer this question, I traced two applications (macrobenchmarks) on Ext3 to generate two sets of block traces; the actual and the actual with locality optimizations destroyed but meaning preserved. Ext3 is a contemporary file system that incorporates advances such as journaling, clustering, and metadata grouping. I ran both sets of traces on a storage system simulator, varying system scale and the virtualization parameters. Sequential and random microbenchmark experiments give insight into how a randomized access pattern can outperform sequential and also stabilize performance.

All the trace generation experiments were run on the same Linux 2.6.11 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 microprocessor, 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. A separate disk was used for trace logging.

I wrote in-kernel code to trace the sector number, size, and I/O type of block operations. Trace records are stored in a circular kernel buffer. A user-level utility polls the buffer, extracting records and appending them to a file on an untraced device. I traced two benchmarks; postmark and build.

Postmark [4] 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.

Build is a build of the kernel and modules of a preconfigured Linux 2.6.11. It is a real workload, not a synthetic benchmark. Both postmark and build start with a mount so that the buffer cache is cold.

The original postmark and build traces were generated from running their benchmarks on an Ext3 file system. I refer to these two original traces as the *smartypants* traces because Ext3 is quite clever about laying out blocks on disk.

I generated *stupid* traces by applying a random permutation of the block address space to the smartypants traces. This maintains the meaning of blocks while destroying the careful spatial locality of a smartypants file system.

Figure 2 shows the breakdown of the block traces. One result of this paper is that the stupid traces can have competitive or even better performance. This is surprising when we look at this figure and contemplate just how deeply I brutalized smartypants to generate stupid. Not only are the I/Os of stupid scattered all over the place with absolutely no regard for interblock locality, but there are many more of them. There are many more of them because stupid only does I/O in units equal to the file system block size of 8 sectors. The workloads are dominated by writes at the block level. The read to write ratio here does not represent the ratio issued by the application because the buffer cache absorbs reads and writes.

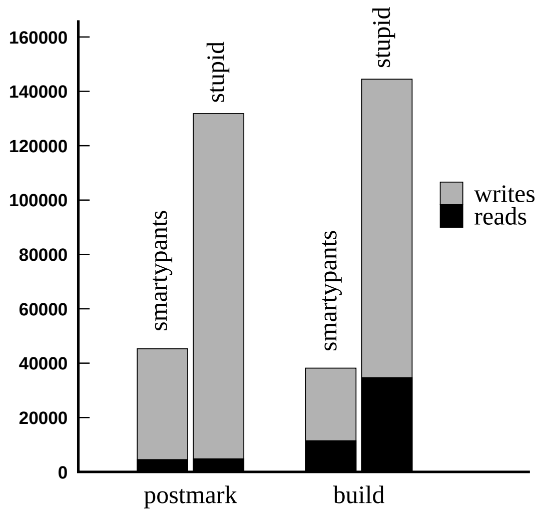

Figure 2: **Breakdown of Trace I/Os**
This figure shows the total number of I/Os for the 4 traces used in this paper. The figure breaks these totals down into their read and write categories. All stupid I/Os are done in file system blocks of 8 sector units.

Throughout this paper, a sector is 512B and a file system block is 8 sectors (4KB). The ratio of a particular I/O type's stupid to smartypants bar height represents the average I/O size of that smartypants I/O type measured in file system blocks. This is because stupid issues I/Os only in the size of file system blocks. For example, smartypants postmark reads are on average approximately equal to the size of a file system block, while smartypants build reads average over two file system blocks.

All experimental approaches to evaluating computer 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 does not vary depending on the behavior of

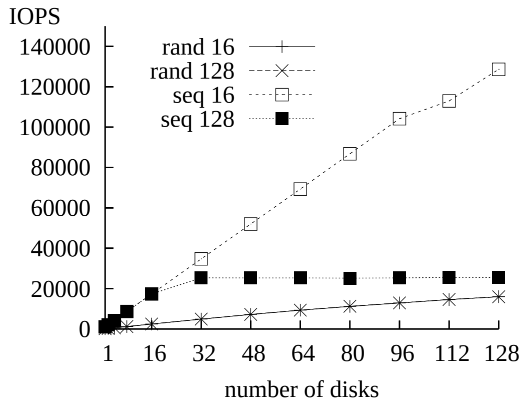

Figure 3: **sequential and random reads**

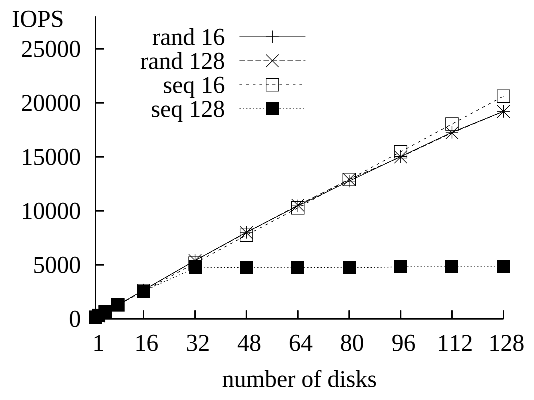

Figure 4: **sequential and random writes**

the system. On the other hand, its central strength is that it represents something meaningful. A recent report by Zhu et al. discusses these issues [9].

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

I used a storage simulator for two reasons. First, it allowed me to experiment with systems larger than those in our laboratory. Second, it eased the exploration of the virtualization parameter space.

The simulator implements a simple operating system for queueing I/Os. It sends the trace requests to the storage system as fast as it can, but with a window of 200 I/Os. A window size of 1 would allow no parallelism while an infinite window would neglect all interblock dependencies. Using a window size between 1 and infinity allows some I/O asynchrony without tracing interblock dependencies and without wildly overstating the opportunities for parallelism.

4 Experimental Results

The microbenchmarks are simple access patterns running directly on top of the simulator. The macrobenchmarks are all generated using the trace-driven simulation approach described in the previous section.

4.1 Microbenchmarks

The 4 microbenchmarks are read or write access with sequential or random patterns. These were run on RAID-0 arrays of varying size. The sequential read and write benchmarks issue I/Os sequentially across the block address space. The random read and write benchmarks issue I/Os randomly across the block address space. In every benchmark, I/Os are done to the storage system in 8 sector units. The microbenchmarks were run for different stripe units. Figures 3 and 4 show the results. The numbers in the figure keys are the stripe units.

Consider the read results of figure 3. Sequential far outperforms random for the smaller stripe unit of 16 sectors. 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. As the stripe unit increases, the benefit of the track caching becomes less and less of a component, bringing the sequential throughput down to the random throughput.

Now consider the write results of figure 4. Writes do not benefit from track caching. Without the track caching, sequential and random writes have similar performance for small stripe units across 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 in both microbenchmarks. These results show how performance can be stabilized across different levels of virtualization by removing spatial locality from the I/O stream. Additionally, these results show that sometimes random access can outperform sequential by balancing load and facilitating parallelism.

4.2 Macrobenchmarks

In this section, I will discuss the results of running the traces on 3 systems; a single disk varying chunk size, an array varying the number of disks and stripe unit, and a volume varying the number of disks, the stripe unit, and the chunk size.

Figure 5 shows the four traces on a single disk. The y-axis is normalized time. The build results are normal-

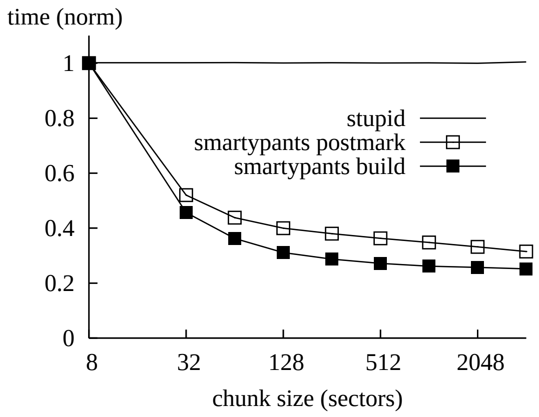

Figure 5: **Build and postmark on a single disk**
The time of postmark and build are normalized separately to the time of their stupid run for a chunk size of 8 sectors. Stupid postmark and build are indistinguishable (varying at most 1.7% for build and 0.4% for postmark) and are shown with one line.

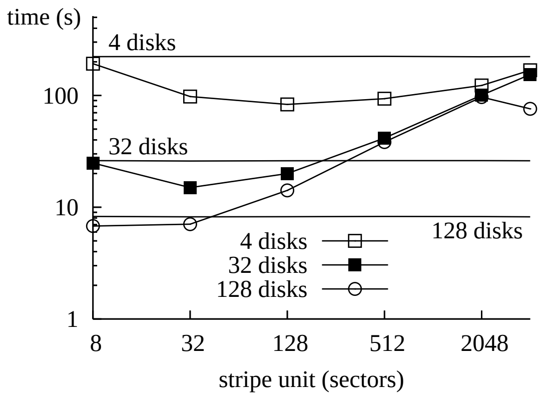

Figure 6: **Postmark on disk arrays**
The stupid results are indistinguishable for a given array size (varying at most 0.9%) and are shown with one line for each array size. The 3 smartypants results are shown as lines with points. The key shows their array sizes.

ized to the time of stupid with a chunk size of 8 sectors. Similarly, the postmark results are normalized to their stupid time with a chunk of 8. Both of the stupid lines do not vary enough to appear independent on this chart, so only one line is shown. As the chunk size gets smaller, the granularity of the virtualization remapping becomes finer, destroying the correspondence between locality in the block address space and physical locality on the disk. When the chunk is equal to the file system block size, the stupid and smartypants traces perform the same. As the chunk size increases, locality in the virtual address space begins to correspond to locality on disk across larger and larger extents. The assumptions of the smartypants optimizations begin to be true and the performance of the smartypants traces both improve by over 60% by a chunk size of 2048 sectors. This shows that all that work on improving local file system performance was not for nothing.

The stability of stupid is not limited to the single disk. You will see this in the array and volume results. Here, however, stupid is worse than smartypants. When a file system is composed into a hierarchical system its stability contributes to the total system stability. A system that values stability over performance might even prefer the stupid approach for a single disk.

Figure 6 shows the performance of the postmark traces on a RAID-0 array varying the number of disks and the stripe unit. The performance of stupid is stable across the stripe units for all 3 array sizes. Stupid scales better than smartypants. For the smaller array of 4 disks, smartypants beats stupid across all of the experimental stripe unit values. By 32 disks, stupid is beating smartypants for stripe units greater than 128 sectors. By 128 disks, smartypants performs better only for the stripe units of 8 and 32 sectors.

The down and up curve of 4 disk smartypants is seen repeatedly in the array and volume experiments. As the stripe unit increases, smartypants benefits from more efficient sequential I/O. As the stripe unit increases even further, the locality clustering of smartypants creates convoying from disk to disk as smartypants jumps from one cluster to another, bottlenecked on the internal performance of some overloaded disks, while some others remain idle. I ran the same set of experiments with the build traces and the result pattern similar.

Figure 7 shows the performance of build on a 128 disk volume that remaps chunks of a RAID-0 array using the volume model of figure 1. The experiments vary the chunk size and stripe unit across 6 stripe units and 9 chunk sizes. The 54 stupid points form an approximate flat plane with stable performance. This plane is shown here as a line. Stupid outperforms smartypants for all those configurations with stripe unit and chunk size strictly greater than 128 sectors. In this set of experiments, smartypants curves down and up across both chunk size and stripe unit. The curve breaks off into stability whenever the independent parameter exceeds the fixed one. That is because the smaller virtualization parameter dominates the remapping and neutralizes the larger one. I ran the same set of experiments with the postmark traces and the result pattern was similar.

Continuing to look at figure 7, consider the smart build trace with stripe of 32 sectors and chunk of 512 sectors. The system processes this trace at an average rate of 115.98 IOPS per disk with that average computed using a sampling period of 10ms. This average has a

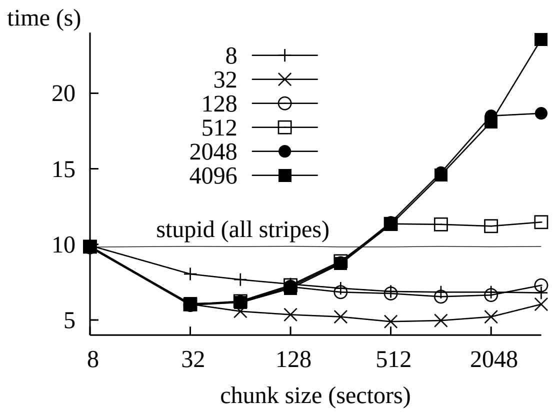

Figure 7: **Build on a 128 disk volume**
Each line varies the chunk size for a fixed stripe unit. Again, the stupid results are indistinguishable across both stripe unit and chunk size (varying 2% across all 54 points). This plane is shown as a line. The 6 smartypants lines are shown with points and the key shows the stripe unit for each.

standard deviation of 9.30% across the 128 disks. The stupid trace on the same system runs at an average rate of 135.11 IOPS per disk with a standard deviation of 7.54% across disks. As you know from figure 2, the stupid build trace consists of over 100,000 more I/Os than the smartypants build trace. The full trace takes longer even though the stupid system is able to process marginally more IOPS per disk and balance load more smoothly. For both benchmarks, the average size of the stupid I/Os is smaller than that of smartypants. Continuing to look at figure 7, now consider the smart trace with stripe and chunk of 4096 sectors. The system processes this trace at an average crawl of 7.05 IOPS per disk with a mammoth standard deviation of 58.42% across disks. During many sampling periods some disks were completely idle while others were overloaded. The stupid trace on the same system runs at an average rate of 133.82 IOPS per disk with a standard deviation of 7.60%. In this case, even though the stupid trace is much longer, it outperforms the smartypants trace. The large standard deviation of smartypants shows how smartypants layout can outsmart itself and defeat parallelism by creating overloaded hotspots.

5 Conclusions

The random permutation of the stupid traces scramble and destroy the careful block proximity decisions of smartypants. From the perspective of smartypants, virtualization also acts as a destructive permutation, though less thoroughly and with greater structure than stupid. Storage virtualization facilitates scalability, fault-tolerance, and reconfiguration, and is therefore unlikely to go away. This paper gives you two take-away results. First, I have shown that under some workloads and virtualization parameters, random layout can outperform the careful layout of a file system such as Ext3. In some cases, random layout can help a system benefit from disk parallelism by smoothly balancing load. The second, and I believe more interesting, result is how differently stupid and smartypants respond to varying virtualization parameters. In these experiments, stupid is always stoically stable while smartypants fluctuates hysterically. Data and file system images can long outlive their storage system homes. I propose random layout as a technique to immunize file systems from the instabilities of storage system configuration.

6 Acknowledgments

Discussion with Margo Seltzer inspired this work. Discussion with Daniel Ellard refined and focused it. Computer scientists at the University of Quebec at Chicoutimi (UQAC) strengthened it by hosting me twice to give public lectures about it.

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, November 1994), pp. 49–60.

[3] Hagmann, R. B. Reimplementing the cedar file system using logging and group commit. In *Proceedings of the eleventh ACM Symposium on Operating Systems Principles* (Austin, TX, USA, November 1987), pp. 155–162.

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

[5] 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), 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 1988), pp. 109–116.

[7] Rosenblum, M., and Ousterhout, J. K. The design and implementation of a log-structured file system. In *ACM Transactions on Computer Systems* (Feb. 1992), vol. 10, pp. 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 *Proceedings of the 1993 Winter USENIX* (San Diego, CA, USA, January 1993), pp. 307–326.

[9] Zhu, N., Chen, J., Chiueh, T., and Ellard, D. Scalable and accurate trace replay for file server evaluation. Tech. Rep. TR153, SUNY Stony Brook, December 2004.