Practical Durable Writes for Commodity Disks

Kyle Jamieson and Eugene Shih

Abstract

We consider the problem of how to guarantee *durable writes*: writes that are guaranteed to be committed on disk when the `write` system call returns. All filesystems can provide durable writes, but not without a performance penalty. Often to properly write the file to disk, multiple disk seeks may be required to put the data in the correct locations.

To reduce this performance penalty, our implementation of durable writes relies on the use of *eager writes*: writes that occur at or near the current disk head location. Our techniques for durable writing are performed at the software level, instead of inside proprietary disk drive firmware. The eager writing techniques we use involve software-based disk head prediction algorithms. In this paper, we describe the design of a straightforward implementation of eager writes in the Linux kernel. Specifically, we show how simple modifications to the *ext2* filesystem can support durable writes.

1. Introduction

Conventional filesystems such as FFS perform durable writes slower relative to asynchronous writes because disk head seeks to various metadata and data blocks are required. In most cases, on a `write` system call, these file systems buffer writes for a few seconds before submitting a group of writes to the disk. Systems such as FFS with SoftUpdates [9] and journaling file systems [15] perform writes in this manner. In effect, these systems trade performance for synchronous semantics: data is not required to be durable on disk when a `write` system call returns.

Some systems, however, would benefit from durable write semantics. NFS [12], database systems, and applications that require frequent `fsyncs` require durable writes to be made frequently.

In the case of NFS servers, a server running NFS prior to version 3 is required to make any file system operation stable on disk before sending a reply back to the client. This is due to the statelessness of an NFS server. As a result, to append a block to a file, an NFS version 2 server must make several time-consuming seeks and disk writes to update both the file's inode and data blocks, before returning a reply.

Database-like applications are another example of systems that have stringent requirements. In order to maintain ACID properties of database transactions in a traditional filesystem, frequent `fsyncs` must be made by the application. Our eager writing technique would yield the same semantics as `write` call followed by a `fsync`, but with a substantial performance improvement. This performance improvement can greatly benefit applications involving web-based e-commerce, where it is necessary to commit to stable storage information pertaining to an order (including credit card information and customer address information), before replying to the user with an order confirmation.

Another application that will require frequent fsyncs is the UNIX `syslog` daemon. In general, the `syslog` daemon does not want to lose data in the case of a system crash (which may be very likely given that a system administrator is interested in its logs). The `sendmail` program also has to be careful about frequently committing sent email to stable storage.

The focus of our paper is to provide these critical applications with the best performance possible. In this paper, we make several contributions. We explore how to perform durable writes efficiently. We use a previously-described technique, *eager writing*, and adapt it in the context of a real file system to perform durable writes with good performance. In order to reduce the latency incurred by conventional implementation of durable writes, we forgo the seeks required and instead perform the write to a contiguous sequence of free disk blocks that are near the current position of the head. Unlike previous work that utilized the eager writing techniques, we show how such a system can be implemented *entirely in software*.

2. Related Work

Improving file system write efficiency has been a topic of much research in the early 1990s. Most of the previous work differs from our resesearch either in terms of how writes are handled, how much hardware support is required, or in the semantics of a `write` system call.

The Log-Based File System (LFS) [10] keeps all file system data in an on-disk log data structure. Consequently, all writes are performed contiguously and without seeks, if all reads are served by the buffer cache. Unfortunately, read costs and log cleaning costs reduce the performance of LFS to the point where it may underperform FFS-like file systems [13, 3]. Furthermore, LFS batches writes, making some system calls (such as rename or write) non-durable. This weakens its durability semantics to the point that it is not suitable for some types of applications.

VLFS [16] is a proposal for an eager-writing file system that is based on the LFS work. VLFS has similar goals to the present work: minimizing the latency of small synchronous writes to disk. Although it is a convincing design, the authors have no practical implementation of their work. Indeed, as an implementation path, the authors propose either enhancing the host-disk interface to include disk-head location information, or migrating the file system into the disk. We differentiate ourselves from the VLFS work by implementing a practical system.

Mime [5] and Loge [6] are disk controllers that organize data independently of the overlying file system. The disk controller assigns new blocks primarily based on the current location of the disk head. The authors also propose several interesting block allocation heuristics. Mime adds a number of features to Loge, including the ability for the application to specify a variety of write semantics as well as faster recovery. The main distinguishing point between our work and Loge/Mime is that we implement eager writing on commodity disks and disk controllers, in software.

Another technique to improve the performance of small synchronous writes is to use non-volatile RAM (NVRAM). Many commercial products [14] and research proposals [1, 2] utilize NVRAM for this purpose. Research has shown that NVRAM can reduce the number of disk writes and the latency of those disk writes. However, as the SPARCstorage product manual [14] points out, NVRAM adds overhead to the disk subsystem, reducing the maximum write throughput of the system. Furthermore, in the event of a disk drive failure, NVRAM will mask the failure, and the `write` (or `fsync`, or `rename`) system call will return with success. If an application depends on the durability of such system calls, an NVRAM-based system might break that application. In that case, the SPARCstorage product displays an error message on the system console. Sun recommends that its users use mirroring or RAID-5 in addition to NVRAM. While this will indeed yield a high-performance, robust system, we believe our approach is capable of providing the same guarantees and reasonable performance with substantially simpler disks and disk controllers.

2.1 NFS

In the specification of NFS version 2, servers are required to perform all file operations synchronously before replying to the client. Through the use of eager writes, NFS version 2 clients would see a marked increase in speed. Version 3 of the NFS protocol, on the other hand, improved write performance considerably by avoiding synchronous writes using a number of different techniques. For example, the NFS version 3 protocol [4] allows servers to respond immediately to the client with a message specifying one of three conditions: data and metadata are not yet stable on the NFS server, some metadata are not yet stable on the NFS server, or data and metadata are both stable on the NFS server. A client that cares about durability must continue to cache unstable data, and will send a `commit` NFS version 3 protocol message to the server when it needs durable-write semantics.

While this technique can improve performance for some applications, those that require frequent fsyncs or saturate the write bandwidth of the underlying file system can benefit from an eager-writing strategy. In addition, NFS version 3 does not allow a server to create files asynchronously, however. This is an area where the eager writing technique could also help NFS. Also, eager writing could benefit NFS performance by increasing the speed of synchronous version 2-like NFS writes. Juszczak takes another approach to improving NFS server performance [7]: delay passing a `write` down to the underlying file system by a small number of milliseconds, in the hope that another `write` request will come along, and the two requests can be batched together in one synchronous write. If this were always the case, write throughput would increase. Juszczak's work is complementary to our work. One can imagine a write-batching, eager-writing system that would, under certain circumstances, result in higher throughput than either technique used alone.

There is one other consideration to bear in mind about extant NFS server implementations. In particular, the Linux NFS server implementation defaults to *asynchronous* behavior. This means that to version 2 clients, a Linux NFS server will by default *lie* about `write` operations, claiming that they are stable on disk when they have merely been submitted to the file system for a write.

Before describing our implementaion of eager writes in detail, we were first examine the feasibility of an eager writing system implemented using only software.

3. Feasibility

In our implementation of a durable-writing file system, we do not attempt to make modifications to disk controllers or to the disk interface of existing commodity disks. Instead, we modify the operating system to allow eager writes to be used with almost any off-the-shelf hard disk.

Since the operating system software is isolated from the disk firmware, prediction can be difficult, especially since our prediction algorithms are time-sensitive and thus can be affected by competing operating system processes. Thus, we want to develop accurate *software-based* disk head prediction techniques.

While the task of disk head prediction in software may seem daunting, earlier work by Lumb et. al. [8] suggests that sufficient information can be extracted from SCSI disks to allow successful disk head prediction. Using knowledge about how hard disks can be modelled [11] and some of the results from [8], we have independently developed a set of experiments to verify the feasibility of disk head prediction. In the process, we were also able to extract some key information about the

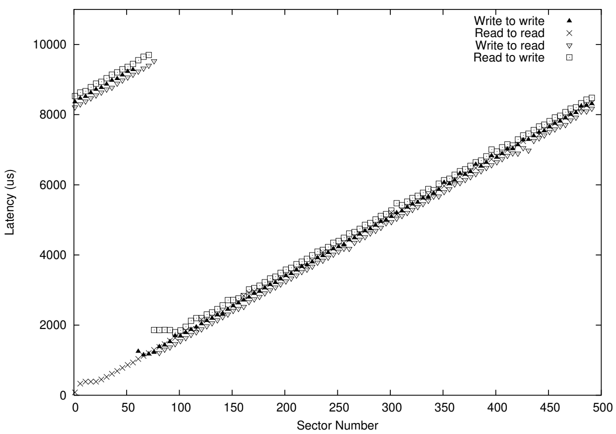

Figure 1: Write latency on $(cylinder, head) = (0, 0)$ as a function of sector number. For each data point, an operation (read/write) to sector zero of the same cylinder on the same head is completed immediately before the read/write to the target sector.

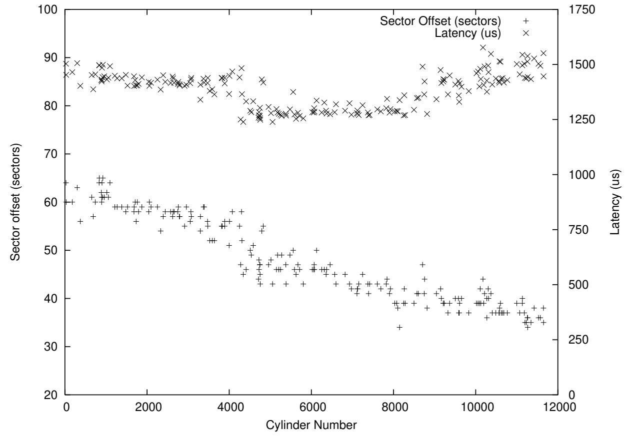

Figure 2: For each cylinder, we determine the sector offset (from a read at sector zero to a write at sector x, for all x) that produces the minimum (over x) write latency. We then simultaneously plot the sector offset and write latency.

behavior of the disk. Unfortunately, a successful eager write implementation needs to be aware of the current location of the disk head.

3.1 Disk Head Prediction Feasibility Experiments

In order to accurately predict disk head location, we must understand the latency of many different disk operations. Some key latency parameters of modern hard disk include head switch latency, settling latency, and rotational latency [11]. Not all of these parameters are well-documented and some are not documented at all. Perhaps the most important parameter is the seek latency. To successfully predict disk head location, the latency of sector to sector seeks of various sizes and types (e.g. read-read, read-write, etc.) must be understood.

For a straightforward implementation of eager writing, seek latency is not as important since we will likely write to a sector on the present track. In this case, no seeks will be required. Instead, we only need to predict the rotational latency of disk operations. However, a more sophisticated eager write scheme may choose to write to adjacent tracks. In that case, predicting the seek latency will be important.

3.1.1 Measuring Rotational Latency

To determine the rotational latency of hard disks, we ran experiments on an off-the-shelf SCSI disk, a Quantum XP32150W, installed in a Pentium 4 machine. In this experiment, the machines were running Linux 2.4.19-16mdk. We used the Linux SCSI generic driver (sg) to obtain direct low-level access to the SCSI disk. All of the experiments in this section use this interface alone.

In Figure 1, we examined the rotational latencies of disk operations on track $(cylinder, head) = (0, 0)$ of our SCSI disk. We measured four different latency parameters. We measured the read-write, read-read, write-read, and write-write latencies. In each case, a read/write was performed to all sectors after a read/write to sector zero.

for C **from** 0 **to** max_cylinders
    do for S **from** 0 **to** max_sectors(C)
        do read physical address $(C, H, 0)$
            write physical address (C, H, S)
            update min write latency and S_{min}

Figure 3: The pseudocode used to determine the minimum rotational delay of the hard disk as a function of cylinder. The results of this experiment are shown in Figure 2.

We will focus mostly on read-write latencies, where a read is performed before a write. However, the discussion that follows will similarly apply for the other three latency parameters. From Figure 1, we can gain some insight about the read-write latency. Around sector 60, we can see a discontinuity at sector number 60. This means that the overhead for a write command to go to the Linux `sg` interface, down to the Linux SCSI subsystem, and through the drive firmware is equal to the rotational delay of 60 sectors for this track. At sector 60, a minimum read-write latency of 1.46 ms is achieved. The fixed overhead for other latencies can also be read off the graph in a similar manner.

The above experiment only examined a single track. Figure 2 shows the minimum read-write latency for each cylinder and the absolute sector for which that latency occurs. The head is randomly chosen for each data point. Figure 3 gives a more precise description of the experiment we ran. This experiment is valuable because it gives us a hint about the read-write latency when the disk head is in a particular cylinder. Given the cylinder, we may be able to predict the location of the disk head by using this minimum read-write latency.

for R **from** 0 to 30
  do pick $C \in_{\text{random}} (0, max\_cylinders)$
    pick $S \in_{\text{random}} (0, max\_sectors(C))$
    pick random head H
    read physical addressp (C, H, S)
    wait R rotations minus command latency time
    write sector S (measure the write latency)

Figure 4: Pseudocode used to determine rotational latency prediction accuracy. Here R represents the number of rotations to wait for, C represents the cylinder we have randomly chosen, and S represents the sector that we have selected, also at random. The results of this experiment are shown in Figure 5.

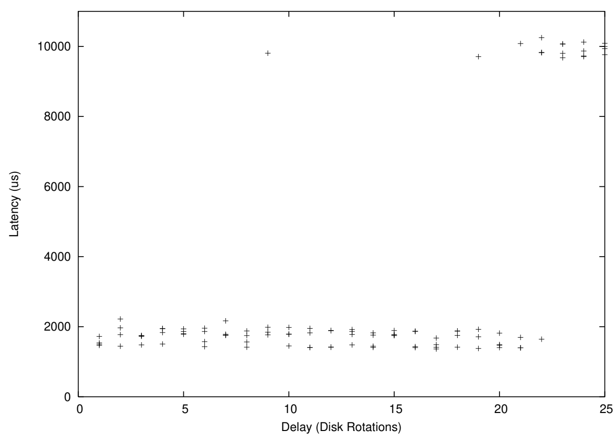

Figure 5: Write latency as a function of disk rotations. For each data point, a read to a randomly-chosen sector on a randomly-chosen $(head, cylinder)$ pair is made. Then after R disk rotations (timed by our software), we write to the same sector and time the write latency. Compare with Figure 1.

3.1.2 Predicting Rotational Latency

Once we understand how to measure the rotational latency, we wanted to determine whether or not we could accurately predict the rotational position of the disk. We designed an experiment to test our accuracy. An outline of the code that we wrote for our experiment is shown in Figure 4. Essentially, we choose a random cylinder, sector, and head and read the sector at that physical disk address. We then wait the amount of time, t_R, predicted for R rotations to take place. Immediately following that time, we write to the same sector S. If the latency of the resulting write is minimal, then our prediction of t_R was correct. From Figure 5, you can see that for $R < 20$, the read-write latency is equal to 1.6 to 2.1 ms, the minimal read-write latency achievable via the `sg` interface (shown in Figure 1).

4. Disk Mapping

In addition to being able to predict rotational latency, we also must be able to map between a physical and a logical view of the disk. In general, this mapping is hidden from the operating system, and must be inferred by timing operations, or bus-specific (e.g. SCSI) functions. A disk head predictor tries to determine the physical sector over which the disk head is currently located. In order to be meaningful to the filesystem, physical sectors needs to be translated to logical block address using a *disk map*.

The basic goal of the disk map is to completey translate logical block numbers (LBN's) to physical sector addresses (cylinder, head, sector). A naive way to accomplish this is to enumerate all user-accessible LBN's. Unfortunately, the storage requirements for such a disk map become prohibitively expensive as the size of the hard disk grows. Moreover, since hard drives will remap logical blocks as physical sectors become defective, the hard disk will periodically need to be remapped in order to keep the disk map updated. Because of these disadvantages, we took a slightly different approach to providing the necessary translation.

4.1 Our Disk Mapping Approach

The authors of [17] show how it is possible to translate physical sectors into logical block addresses and vice versa without using a complete disk map. The authors describe algorithms to determine the data layout on a drive. Unfortunately, the code they used is not available to the public. Therefore, we developed our own disk mapping algorithm, based on a description in their paper.

In this approach, instead of recording every possible translation, we only record where tracks begin and end. Recording track boundaries is sufficient for translating logical block numbers to disk addresses if we assume that all available sectors on a given $(cylinder, head)$ pair are mapped *before* proceeding to the next cylinder and/or head. In addition, we assume physical sector zero is present on each $(cylinder, head)$ pair. Both of these assumptions were true for the disks we used in our experiments and implementation.

To determine the track boundaries for our disks, we used the following algorithm. First, make an initial guess of the number of sectors per track S by scanning from logical block zero onward until we reach a new head or cylinder.

During the scanning process, we translate logical addresses to physical addresses and record the track boundaries as we go. To speed the mapping process, we only translate logical addresses of the form kS where $k > 0$. If kS translates to a sector larger than zero on the next $(cylinder, head)$ pair, then a *slipped defect* must be present. This implies that the logical addresses in $[(k-1)S, kS]$ would have mapped to physical sectors that contain defects. Consequently, the drive "slips" the mapping by one for each problematic physical address. In this case, our scanning algorithm will look either up or down to find the logical address that maps to sector zero of the next $(cylinder, head)$ pair. Track boundaries are marked each time we find sector zero of each $(cylinder, head)$ pair.

In many disks, S does not stay constant over all $(cylinder, head)$ pairs. Some drives have more than one *zone* (also called *notch*), a range of $(cylinder, head)$ pairs over which S does stay constant. We detect a new zone by noticing if our prediction of sector zero is off by a large constant factor in consecutive translations. We then increase or decrease our guess of S and continue mapping.

5. System Design of Eager Durable Writing

As the previous sections show, performing durable, eager writes in the operating system is possible if the disk head location can be predicted accurately. To show the benefits of this technique, we modified parts of the Linux kernel (2.4.18-17). Specifically, we modified the ext2 filesystem and parts of the buffer cache to implement durable writing.

5.1 Implementing Durable Writes

In order to perform fast durable writes, we modified ext2 to write data in contiguous physical sectors starting at the present disk head location. Our implementation modifies how ext2 behaves each time calls to `sys_write()` are made.

In ext2, when a write system call is made, each block[1] of data is allocated to a disk location independently of other blocks. As blocks are assigned locations on disk, buffers in the kernel's buffer cache are also allocated. Typically, this data is not committed to disk immediately. Rather, dirty data in the buffers is committed to disk asynchronously by a system kernel daemon. ext2 is also able to perform synchronous, durable writes. However, the latency of these writes can be large due to ext2's block allocation policy. Since the ext2 block allocation policy favors placing data near their respective inodes and in nearby block groups, it is likely that one or more seeks will be required to commit data to disk synchronously.

In order to make ext2 write eagerly to disk, when a system write call is made, we first determine the total number of blocks, *n_blocks* that will be allocated. need to be allocated. Once we know the total number of blocks that need to be written, we allocate buffers for those blocks. To allocate disk locations for these blocks, we call the procedure *eager_write_message*. *eager_write_message* will return the starting logical block number, *start_block*, of a set of free contiguous blocks. The buffers are then assigned sequential block numbers in an increasing order. Once the buffers have been assigned logical block numbers, we synchronously commit all of them to disk. If the prediction and block allocation is accurate, then no seeks or head switches will need to be performed when the disk writes the data.

While the allocation policy we employ is a fairly simplistic one, as the number of files on a disk increases, the allocation begin to fail since long sequences of free blocks will become more difficult to find. If our procedure is unable to find a satisfactory disk location, then we fall back onto ext2's default allocation policy, that is, block-by-block allocation.

5.2 Metadata and Indirect Blocks

As blocks are allocated in the buffer cache, the buffer corresponding to the inode of the file is modified to reflect a change in the file size and to link data blocks to the inode via indirect blocks. If the inode is *not* in memory, then a disk operation to read the inode will be issued. This will affect the performance of the synchronous write. Once the inode data is changed in memory, it will need to be flushed to disk. However, we do not immediately flush the inode data eagerly to disk. This is because inodes are pre-allocated to disk locations when the disk is created and thus, cannot be allocated eagerly without changing the ext2 filesystem layout substantially. Only data blocks are written asynchronously. Indirect blocks are also written asynchronously.

Because we do not flush dirty inode buffers to disk immediatly, the system as described is at a higher risk for losing data. Though this data loss risk is no worse than that of many existing file systems, not flushing the metadata immediately reduces the durability of the writes. To ensure durability, we create a special metadata block and commit it to disk in front of the data blocks that are being written eagerly. Figure 6 shows the fields that we store in this metadata block. The information can be used to link each of the blocks to the inode by a background process. The `inode_nr` field stores the inode number to which the succeeding blocks belongs to. The `first_block` and `last_block` fields indicate the absolute logical block numbers of the first and last block in the extent of data. The field `c_time` stores the time that the extent was created. Finally, the `inode_dev` variable stores the device number of the inode.

```
struct special_metadata {
    unsigned long inode_nr;
    unsigned long first_block;
    unsigned long last_block;
    unsigned long c_time;
    kdev_t inode_dev;
};
```

Figure 6: Contents of the special metadata block used to improve the durability of file writes.

5.3 Prediction Technique

To obtain the logical block number where we should be writing, we first obtain a hint from the disk prediction daemon. In order to provide an accurate estimate of the disk head location, our file system notifies the prediction algorithm of the logical block that *every* read or write disk operation completed on. Using the disk map discussed in Section 4, we can translate logical block numbers to physical disk addresses.

To provide the estimate, we assume that the disk head stays on the track where the previous disk operation left off. Using precision timing information, we are ablt to determine the phyical sector where the disk head will be located when the next disk operation is submitted.

5.4 Block Allocation

In our simple inefficient implementation of eager write, after a prediction for the disk head location is obtained, one needs to be able to find an appropriately sized extent of free blocks where the data can be written. The ext2 filesystem does not maintain a list of extents indexed by size. As a result, searching for free blocks in ext2 can involve loading many

[1] By default, ext2 has file block sizes of 4096 bytes.

free block bitmaps for a number of different block groups. The impact of this loading can have a negative effect on eager write latency.

5.5 Background Process

The data numbers permits the system daemon to correctly point the inode to the data blocks that were created eagerly. In some cases, indirect blocks will need to be allocated by this background process.. Using the absolute logical block numbers, we can determine whether indirect blocks will need to be allocated.

5.6 Crash Recovery

In the case of a system crash, the only way to ensure that all the data is recovered, a complete disk scan will be necessary since metadata may not have been written to disk. While the time spent performing a disk scan can be large, it is the only way to ensure that all the data stored on disk is recovered and the system is consistent.

After a system crash, our disk recovery tool will first check to see whether or not the disk was unmounted cleanly. If not, it proceeds to scan the entire disk looking specifically for the special metadata blocks we described in Section 5.2. Using the fields stored in this block, the inode responsible for the extent following it can be united with this orphaned group of data as described above.

6. Performance

In this section, we evaluate the performance of our prototype filesystem. We will compare our filesystem to Linux's ext2 filesystem mounted in synchronous mode.

6.1 Write Performance

To measure the raw write performance of our eager writing system, we simply wrote a script to write files of increasing sizes and measured the latency required to completely commit the write to disk. Figure 7 shows the time taken to complete a write from submission to the operation to completion to disk. Note that our numbers reflect raw write performance and do not reflect the additional time required to flush metadata and to allocate indirect blocks in the background.

This result is expected because the eager-writing file system does not have to perform seeks to write data into the correct locations.

6.2 Read Performance

Because data conveyed through write system calls are allocated eagerly, it is possible that the read performance of the files in our system will suffer. However, with increasing buffer caches sizes, this penalty will only be incurred on the first read of a file. Subsequent reads can be served directly from the buffer cache. Furthermore, our system will operate with a cleaner, similar to the LFS, that will gather eagerly-written data into contiguous extents on disk. The main question to be answered in future performance analysis is whether our cleaner will have more or less performance

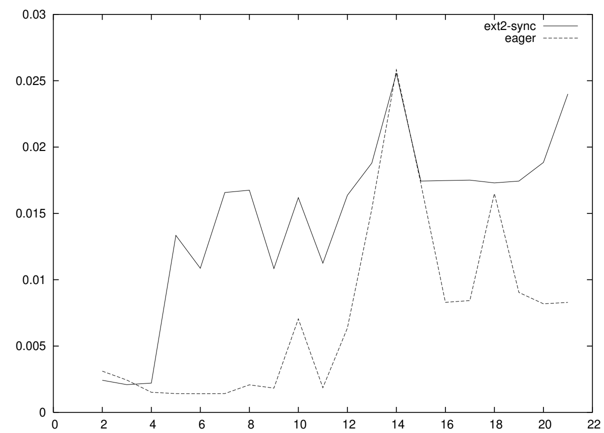

Figure 7: The write latency of ext2 mounted with the synchronous option and our eager writing filesystem for various file sizes. The x-axis shows bytes and the y-axis shows seconds.

impact than the LFS cleaner. We expect that our system can perform better, since data from writes will tend to be placed near data from immediately-previous reads, due the the way our eager-writing algorithm works.

7. Discussion

While our results our positive, we believe that the durable writing performance can be further improved by using a different eager allocation policy. For example, instead of failing the eager allocation when no contiguous free blocks can be found, we might be able to instead use two or more smaller contiguous free blocks to store the data blocks. While this may increase the write penalty, it may not be as bad as falling back to ext2's allocation policy.

In addition, for continuing development, we plan on using FreeBSD 4.x as our base operating system. Moreover, instead of using ext2 as the starting point, we believe that the use of a file system that supports quick searches of free extents, e.g. XFS, will allow eager writes to proceed with lower write latency since multiple free block bitmaps from block groups will not need to be loaded to find an appropriate sized free extend.

8. Conclusion

In this paper, we have shown that eager durable writing is feasible with relatively low latency cost. With our prototype implementation, we showed that eager writing can be performed on an off-the-shelf disk using software techniques alone. We further showed that synchronous writes using eager writing can be implemented with better or comparable performance as ext2.

9. References

[1] Baker, M., Asami, S., Deprit, E., Ousterhout, J., and Seltzer, M. Non-volatile Memory for Fast, Reliable File Systems. In *Proceedings of the 5th International Conference on Architectural Support for Programming*

Languages and Operating System (ASPLOS) (New York, NY, 1992), vol. 27, ACM Press, pp. 10–22.

[2] Biswas, P., Ramakrishnan, K. K., and Towsley, D. Exploting Non-Volatile Memory in Disks for Write Caching. Tech. Rep. UM-CS-1994-067, University of Massachusetts, Amherst, 1994.

[3] Blackwell, T., Harris, J., and Seltzer, M. I. Heuristic Cleaning Algorithms in Log-Structured File Systems. In *Proceedings of the Winter USENIX Technical Conference* (1995), pp. 277–288.

[4] Callaghan, B., Paulowski, B., and Staubach, P. NFS Version 3 Protocol Specification (RFC 1813), June 1995.

[5] Chao, C., English, R., Jacobson, D., Stepanov, A., and Wilkes, J. Mime: a High-Performance Parallel Storage Device with Strong Recovery Guarantees. Tech. Rep. HPL-CSP-92-9 rev. 1, HP Labs, 1992.

[6] English, R. M., and Stepanov, A. A. Loge: a Self-Organizing Disk Controller. In *Proceedings of the Winter USENIX Technical Conference* (1992), pp. 237–251.

[7] Juszczak, C. Improving the Write Performance of an NFS Server. In *Proceedings of the USENIX Winter 1994 Technical Conference* (San Fransisco, CA, USA, 1994), pp. 247–259.

[8] Lumb, C., Schindler, J., and Ganger, G. Freeblock Scheduling Outside of Disk Firmware. In *Proceedings of the Conference on File and Storage Technologies (FAST)* (Monterey, CA, 2002).

[9] McKusick, M. K., and Ganger, G. R. Soft Updates: A Technique for Eliminating Most Synchronous Writes in the Fast Filesystem. In *Proceedings of the FREENIX Track: 1999 USENIX Annual Technical Conference* (June 1999).

[10] Rosenblum, M., and Ousterhout, J. K. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems 10*, 1 (1992), 26–52.

[11] Ruemmler, C., and Wilkes, J. An Introduction to Disk Drive Modeling. *IEEE Computer 27*, 3 (March 1994), 17–29.

[12] Sandberg, R., Goldberg, D., Kleiman, S., Walsh, D., and Lyon, B. Design and implementation of the Sun Network Filesystem. In *Proceedings of the Summer USENIX Technical Conference* (1985), pp. 119–130.

[13] 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 Winter USENIX Technical Conference* (1993), pp. 307–326.

[14] Sun Microsystems. *SPARCstorage Array User's Guide (Part No. 802-2042-11).* Mountain View, CA, USA, September 1996.

[15] Sweeney, A., et al. Scalability in the xfs file system.

[16] Wang, R. Y., Anderson, T. E., and Patterson, D. A. Virtual Log Based File Systems for a Programmable Disk. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI)* (1999), pp. 29–43.

[17] Worthington, B., Ganger, G., Patt, Y., and Wilkes, J. Online Extraction of SCSI Disk Drive Parameters. In *Proceedings of SIGMETRICS* (Ontario, Canada, 1995).