Journaling Filesystem Consistency on Disks Without Tagged Command Queuing

by

Jonathan Pearson

Submitted in Partial Fulfillment

of the

Requirements for the Degree

Master of Science

Supervised by

Kai Shen

Department of Computer Science
The College
Arts and Sciences

University of Rochester
Rochester, New York

2007
Acknowledgments

I wish to thank Kai Shen for always being available to provide help and support when I needed it.

I would also like to thank my wife Sara and my mother, as both unyieldingly pushed me to finish this.
Abstract

A filesystem’s sole purpose is to store data so it can be easily accessed at a later time. Part of that entails recovering properly from a system crash. But a difficulty that modern filesystems face is the advent of write caching in a disk. The disk will report that a write operation has completed before the data is actually secure on the magnetic platter. How is a programmer to respond to this outright lie from the hardware?

The problem goes even further. Journaled filesystems depend on the order of the write operations they send to the disk. If the real data is written before the journal, then there is not only no point in having the journal, but it actually gives a false sense of security. After a crash, the disk checker will only replay the journal, never bothering to examine the real data on the disk for consistency.

There is a solution to this problem. It is called Tagged Command Queuing (TCQ) in the SCSI-2 specification. And it is optional. Filesystems, Ext3[4] in particular in this paper, use TCQ exclusively and have no fallback. We present a fallback solution that depends on a required feature of the SCSI-2 specification, Force Unit Access (FUA). Our results showed that write-intensive workloads display a significant slowdown from the FUA-based solution. However, the performance impact is still less than that incurred by using other methods, such as synchronizing the cache after each journal write.
# Table of Contents

## Acknowledgments

## Abstract

## List of Figures

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Introduction</td>
<td>vi</td>
</tr>
<tr>
<td>2</td>
<td>Background</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>Evaluation</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>Solution</td>
<td>9</td>
</tr>
<tr>
<td>5</td>
<td>Results</td>
<td>11</td>
</tr>
<tr>
<td>6</td>
<td>Related Work</td>
<td>15</td>
</tr>
<tr>
<td>7</td>
<td>Conclusions</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>References</td>
<td>18</td>
</tr>
</tbody>
</table>
List of Figures

3.1 An illustration of how write reordering may affect journal operations. The "Target Zone" is where the data is going to end up. With the head in the position that it is now, if both the journal data and the real data were in the disk cache, the disk would write the real data on its way to the journal. If the disk lost power while writing the real data, there would be no record that changes were being made, as the journal was not updated. ............................ 7

5.1 Our measurement of the performance after the SPECweb99 benchmark had been running long enough to transfer about 1.3 gigabytes. The results were constantly fluctuating as the benchmark ran, so it was not feasible to programmatical ly choose when to start and stop it. You can see that the performance of the SPECweb99 benchmark was not affected by our changes to the filesystem. ............................... 12

5.2 Our measurement of the performance of the TPC-B benchmark with one branch and 99 threads, running for 1000 seconds. A very obvious difference in performance is visible here. ............................... 12
1 Introduction

One of the many responsibilities that a filesystem has is to remain consistent. In this context, “consistency” refers to the ability of a filesystem to always be in a known state, accounting for data and unused space on a storage device. The difficulty with keeping a filesystem consistent, however, lies in the unpredictability of events such as power loss. When the system loses power, there is no way to know what data has made it to permanent storage without taking precautions.

That is where journals come in. Journals are a precaution against sudden interruption. The idea behind a journaled filesystem is that you write a description of any change you are about to make to a “journal,” which is a special area of the disk, before you write the change itself. Then, once you’ve finished the real write operation, you may clear that entry from the journal to indicate that it is safe. If the power fails before the journal is cleared, the filesystem driver will load up and see that there are active journal entries. In that case, it knows exactly what was being written and where it was supposed to go, and the driver can use that data to repair the disk.

But write ordering may not be guaranteed. Modern magnetic disks have caches. A cache is a small block of memory that helps speed operations by holding onto data. That allows the disk to respond to read requests without repositioning the head, and to respond to write requests before actually writing the data to the platter. The disk may even go further than just holding onto dirty (unread) data until the head is in place: It may even change the order of the write operations themselves. Why? If the
head is going to pass near one dirty block on its way to another part of the disk, it is in an opportune place to write the dirty data. With the inclusion of disk caches, the act of writing the journal before the real data becomes a much more complex issue.

In this paper, we investigate the issue of re-ordering using SCSI disks, because of their complete and well-followed specification, and their prevalence in industry. One highly useful piece of the SCSI specification is called Tagged Command Queuing, or TCQ\cite{5} (Pages 63-64). This feature of SCSI storage devices was introduced to combat the problem discussed above. Basically, TCQ allows many write operations to be sent to the disk asynchronously (without waiting for confirmation that the operation is finished). The disk will then send a message back to the commanding system when it has finished the operation. With TCQ, a filesystem can send the journal to disk while doing other operations, and send the data after the disk confirms that the journal has really been written. TCQ, however, is an optional part of the SCSI specification\cite{5} (Page 56). So now the issue becomes how to ensure consistency when TCQ is unavailable.

We present a solution using the Force Unit Access (FUA) bit\cite{6}, and apply our research to the Ext3 filesystem\cite{4}. We chose Ext3 in particular due to the large amount of documentation for it, and the fact that it is open source. In a SCSI read request, setting the FUA bit will force the disk to read from the physical medium, bypassing the cache. Similarly, in a SCSI write request, setting the FUA bit will force the disk to write to the physical medium, and not just write the data to the cache (to be committed at an arbitrary time in the future). This method provides a fine level of control over, and provides more security against inconsistency than simply ignoring the problem.

The rest of this paper is organized as follows: Chapter 2 describes the background of various methods that may be used to guarantee write ordering. Chapter 3 evaluates our interpretation of the problem and investigates issues within the Linux kernel (version 2.6.17.13). Chapter 4 describes our chosen solution. Chapter 5 measures performance of various kernel configurations. Chapter 6 cites related work. Finally, chapter 7 concludes with a summary of our findings.
2 Background

There are four main methods to force write ordering:

1. Tagged Command Queuing
2. Synchronize the cache after each journal write and before each journal clear
3. Force all writes through the cache
4. Force journal writes through the cache and delay journal clear until after a synchronize cache operation

Tagged Command Queuing is the standard method, and is used when available on Ext3. It was introduced in the SCSI-2 specification. It works by giving all write commands a "Tag," which is simply a unique number. When the write makes it to the platter, the drive sends a message back to the host system. When the filesystem driver receives the message, it knows that the journal will survive a power loss and can send the rest of the write.

Synchronizing the cache after each journal write and before each journal clear is not used by the Ext3 filesystem under any circumstances. Forcing a complete cache flush will cause the disk to write more than just the journal. As a result, an inordinate amount of disk wear-and-tear could be caused by the number of times the head needs to move around; the journal will likely be in only one place, but the data can be destined for any region on the disk.
Forcing all writes through the cache will likely slow the disk to at best half its normal speed. Read operations will still benefit from the presence of the cache, however write operations will all be delayed while waiting for the head to move and the disk to rotate to the proper position.

Forcing journal writes through the cache is a much more elegant solution than any but Tagged Command Queuing. Journal writes only make up about half of all writes, so the impact will be significantly lower than that of forcing all writes through the cache. But we must not clear the journal entries until after we know the cache to be synchronized; otherwise there is no way to know when the real data has hit the physical medium and it is safe to remove it from the journal. The most efficient time to synchronize the cache and clear the journal would be once the journal is full, as that would result in the fewest cache synchronizations. Additionally, the journal clears could probably be chained together into a single write operation, further benefiting from the lazy clear. Under Ext3, all data is journaled by default, not just metadata, so forcing all journal writes past the cache will provide the same benefit to user data as to the filesystem itself.
3 Evaluation

To confirm that our research would not be in vain, we needed to make sure that the FUA bit was not being set while writing to a disk that did not support Tagged Command Queuing. However, TCQ is a common feature, and there is no easy way to tell if the disk you are using supports it, so we tricked the kernel. With one small change to the SCSI driver, we were able to make it appear that TCQ was not supported by any SCSI disk plugged into the system. In particular, by doing so, filesystems wanting to use it for journaling would need to find another way to make sure that their data was making it to disk.

To be more specific, within the drivers/scsi directory of the kernel source (version 2.6.17.13) is a file called “scsi_scan.c.” This file, around line 725, contains code that detects whether Tagged Command Queuing is supported by the SCSI device it is scanning. By replacing the section that attempts to detect the feature with the line “sdev− >tagged_supported = 0;” the kernel will believe that TCQ is never supported.

The results were surprising. Of the five filesystems we tested (Ext2, Ext3, JFS, ReiserFS, and XFS) only XFS ever set the FUA bit. A bit of further investigation revealed that even when TCQ is enabled, XFS continues to use the FUA bit. We speculate that this is because the FUA bit was available in the first version of the SCSI specification, whereas TCQ was not. XFS was introduced in 1994[16], and the SCSI-2 specification was introduced in September of 1993[5], so it is reasonable to assume that most of the development of XFS occurred before TCQ was even documented, much less
Another method, besides for TCQ or the FUA bit, that a filesystem can use to be certain of data making it to the disk, is to tell the disk to synchronize its cache. The SCSI-2 specification supports a Synchronize Cache command\[5\] (Page 164) for exactly that purpose. The problem with using it, however, is its broad scope. Synchronizing the cache affects more than just the most recent write; in the worst case, it can force the disk to move its head across the entire platter, if there are dirty blocks at both ends. It would be like putting a fresh set of sheets in the wash because a piece of dust landed on them: It is serious overkill.

To determine whether the Ext3 filesystem uses the Synchronize Cache command to force journal writes, we performed the TPC-B database benchmark\[15\] (more on TPC-B later) on the system using both an out-of-the-box kernel as a control, and a kernel that had been modified to see all SCSI disks as not supporting TCQ. Our hypothesis was that the benchmark would be worse on the modified kernel if Ext3 used Synchronize Cache commands (due to the larger amount of head seeks that would be necessary to flush the cache), or at least equally well if it was not being used. Our results showed that the performance of the benchmark without TCQ support was just slightly higher than with it. Therefore, Ext3 must make no effort to guarantee write-ordering when the disk does not support TCQ, which, as we said, is an optional disk feature.

Beyond determining whether any method beyond TCQ is used by Ext3 to guarantee write ordering of the journal, we also needed to prove, at least theoretically, that write ordering can cause concrete problems. To show this, imagine that a program makes the proper system calls to write a few megabytes to a new file (initially empty) on disk. To fulfill this request, the Ext3 filesystem must create both metadata blocks (Ext3 uses a list of direct links to data blocks, followed by a link to a block containing direct links to data blocks, then a double-indirect link, and finally a triple-indirect link; therefore when the user wants to write more than about 48 kilobytes, a new single-indirect block must be created, and if the user is writing more than about 4 megabytes, a double-indirect and some corresponding single-indirect blocks\[3\]).

When making any changes to the data in the filesystem, the data is first written to
Figure 3.1: An illustration of how write reordering may affect journal operations. The “Target Zone” is where the data is going to end up. With the head in the position that it is now, if both the journal data and the real data were in the disk cache, the disk would write the real data on its way to the journal. If the disk lost power while writing the real data, there would be no record that changes were being made, as the journal was not updated.

the journal and then to the disk itself. So let’s say that all of these write operations are sent to the disk, which caches them and reports back success. The journal writes were sent first, but that does not mean that they will reach the platter first. Now, the user has a sudden burst of bad luck: The disk head was nearer to the file’s main INode than to the free block list, the journal, or the newly allocated indirect blocks. The disk, of course, decides to get the dirty data out of the cache, and updates the INode to point to the new indirect blocks. The free list still needs to be updated, the indirect blocks need to have fresh data written to them, and the journal has not been written yet. Now the power fails, and there is no backup attached to the system, so dirty data in the disk’s cache is lost. When the power comes back on and the machine is rebooted, fsck sees that the disks was not unmounted cleanly, so it replays the journal. The journal, however, does not mention the last change that was written to disk. The user goes about his business, and tries to look at the file he created just before the power failure. There are a number of possibilities here, especially if the new blocks contained uninitialized data. The two worst possibilities are these:

1. The filesystem could try to dereference random pointers and crash the kernel
2. The user could see that the file doesn’t look right and delete it; the filesystem dereferences the uninitialized pointers in the new blocks and deletes data belonging to other files, even going so far as to place those blocks in the free list.

The first situation is probably better, as the user may have a chance of grasping the essence of the problem and run a complete check. But in the second situation, the filesystem is silently and slowly destroying itself. The user may expand an existing file, which then claims the blocks which were incorrectly added to the free list. He then may delete the file that those blocks should belong to, propagating the cycle of destruction. This is, of course, both a horrible and an exaggerated situation, but it illustrates how serious the write ordering problem is. Edmund B. Nightingale, et al. come to a similar conclusion in their paper “Rethink the Sync”[10].
4 Solution

To continue our experiment, we decided to modify the Ext3 filesystem driver (in a very naive way) to use the FUA bit when writing to the journal. We did not go so far as to delay journal clears and synchronize the cache immediately before, but this should provide a reasonable approximation of the results one would find by implementing the full requirement. The reason for the similar results is due to a tradeoff – the journal is both written and cleared for every write, which means that there are more (synchronized) journal operations going on than there would be with the delayed clear functionality. On the other hand, there is no cache synchronization going on.

As mentioned above, the Synchronize Cache command was another option, but it does not give the fine-grained control that the FUA bit offers. Additionally, using the Synchronize Cache command after every journal write would require either the journaling code to know what kind of disk to which it was writing (so that it could know how to issue a synch as necessary), or it would require the SCSI driver to be able to differentiate journal writes from normal data. In either case, the resulting solution would violate guidelines encouraging the separation of interfaces, and it would make maintenance on both the SCSI driver and the journal code more complex. Beyond that, the performance hit would be significant, as would the wear-and-tear on the mechanical parts of the disk, as mentioned above.

Implementing the FUA bit solution is a much simpler prospect. The kernel structures dealing with generalized disk access have fields governing what kinds of service are
requested, including whether direct media access should be forced. Our modification consisted of less than ten lines of code. As journal writes were already being passed around as synchronous operations, we only needed to look for that (as opposed to simple writes) and set the correct option when the messages were passed to the next lower driver. Voila, an Ext3 system with a FUA-enforced synchronous journal.

Specifically, under the “fs” directory in the kernel source (version 2.6.17.13) is a file named “buffer.c” containing the relevant code. Our change involved two separate functions, “submit_bh” and “ll_rw_block.” In submit_bh, we watch for SWRITE or WRITE_BARRIER to be passed as one of the parameters (as opposed to WRITE). When that is the case, we pass BIO_RW_SYNC as one of the flags in bio->bi_rw. In the normal kernel, however, SWRITE is never passed to submit_bh, which is why our other modification is necessary. The function ll_rw_block normally passes WRITE to submit_bh, even when it is called with SWRITE itself. We modified it to simply pass what it received on to the next level.

To be certain that these changes performed correctly, we made these modifications to our instrumented kernel, which watches for FUA writes on the SCSI bus. That modification was mostly made in the “elv_insert” function of “block/elevator.c” with some help from a couple of new system calls. When a request is being inserted and is directed at the SCSI bus, we check whether it is a write and whether it has the FUA bit set. Counters exist to track both total writes and forced writes, and custom system calls return those counters.

With the modifications made to force FUA usage on journal writes in place, we simply used our new system calls to see if the FUA bit was being set. Our hypothesis was that approximately 50% of all writes to the SCSI disk would have the FUA bit set, and our tests revealed that the number was generally around 49.6%, close enough to be considered a positive result. The lesser number could be from write combining within the kernel, since the journal is much more likely to all be in one contiguous block than a normal file elsewhere on the disk. Basically, a write that spans two disparate data blocks in a file may be written to the journal as two contiguous data blocks, making the overall write counter increase slightly faster than the FUA write counter.
5 Results

To determine the performance impact of forcing journal writes directly to disk, we ran two benchmarks. One of them, SPECweb99[13], was chosen to make sure that read performance was not affected by our change. We implemented the static portion of SPECweb99[13] (Section 4.5), which simulates a web server (in this case Apache[1]) serving requests for files of various sizes between 100 and 1,000,000 bytes. The benchmark measures the number of simultaneous connections the machine can handle before the rate of dropped connections increases significantly.

The other benchmark we ran was TPC-B[15]. TPC-B simulates a bank with branches, tellers, and customer accounts. It measures the number of transactions per second that can be performed against a database, in this case MySQL[9]. A transaction consists of a user account, teller, and branch balance all being altered by the same amount (simulating a customer depositing or withdrawing money through a teller at a bank branch), while at the same time adding a log entry of what was performed into a history table.

The static portion of SPECweb99 measures read throughput only. We chose it to confirm both that our modification indeed only affected disk writes, and that disk reads were not somehow adversely affected. Figure 5.1 demonstrates that these criteria were met, as the server performed equally well independent of which filesystem was in use.

In Figure 5.2, you can see that write performance definitely takes a hit after the modification to the filesystem. TCQ is much more efficient than using the FUA bit. But TCQ is an optional portion of the SCSI specification, so there should be an alternative
CHAPTER 5. RESULTS

Figure 5.1: Our measurement of the performance after the SPECweb99 benchmark had been running long enough to transfer about 1.3 gigabytes. The results were constantly fluctuating as the benchmark ran, so it was not feasible to programmatically choose when to start and stop it. You can see that the performance of the SPECweb99 benchmark was not affected by our changes to the filesystem.

Figure 5.2: Our measurement of the performance of the TPC-B benchmark with one branch and 99 threads, running for 1000 seconds. A very obvious difference in performance is visible here.
when it is unavailable. The FUA bit is required, though, so it is a reasonable fallback for a journaled filesystem to use when necessary.

In Figure 5.2, you can also see that with TCQ disabled, Ext3 does not fall back on any other method to guarantee write ordering. (Although one could argue that it may provide evidence for the fallback being faster than TCQ, why would it be a fallback instead of the standard method?) Therefore, there is a real need for alternate methods, as Ext3 is a very popular filesystem running on diverse hardware.

For comparison, we also ran the TPC-B benchmark on an unmodified kernel with the disk mounted synchronously. The results were inadmissible by the TPC-B specification, as more than 10% of the database transactions took more than two seconds. More precisely, 100% of the database transactions took more than two seconds. Additionally, had the results been admissible, the numbers were beyond bad. The normal kernel averaged almost 290, the experimental kernel averaged about 165; the normal kernel with a disk mounted synchronously scored an average of about 6.3.

Mounting a disk synchronously tells the kernel not to buffer any data, but to send everything to disk and wait for it to respond that it has been written. However, this does not guarantee write ordering any more than mounting the disk asynchronously. This is unfortunate, as an uninformed user may believe that a synchronously mounted filesystem is completely safe. In reality, it only reduces the chances of write ordering causing a real problem because, as writing takes so much longer, the disk head has more time to move around and write the dirty cache blocks.

We also attempted to generate a kernel that would use the FUA bit to force all writes, however this proved too great a challenge. Not having physical access to the machine meant that it was impossible to view the output as it tried to boot the new kernel, and all we saw was that Grub used the fallback choice every time. Our estimate, however, would be that it would score approximately 82 on the TPC-B benchmark. This calculation is based on the inverse relationship between the percentage of FUA writes and the TPC-B rating. The formula would be:
\[ \text{old\%} \times \text{oldTPCB} = \text{new\%} \times \text{newTPCB} \]

\[ 49.6\% \times 165.2 = 100\% \times \text{newTPCB} \]

\[ (49.6\% \times 165.2)/100\% = \text{newTPCB} \]

\[ (49.6\% \times 165.2)/100\% = 81.94 \]

49.6\% is the percentage of FUA writes, mentioned above. 165.2 is the current average TPC-B score.
6 Related Work

To the best of our knowledge, there has been no other formal work in the area of journaling in the presence of write reordering. The general assumption seems to be that everything supports Tagged Command Queuing, and that if it doesn’t, then the data must not be important.

However, there are a number of works in related areas. The paper Rethink the Sync[10] includes a short section mentioning that Ext3 does not guarantee persistence without write barriers. According to a short paper entitled Disks from the Perspective of a File System[8], the SATA standard includes something similar to TCQ. Exploring Failure Transparency and the Limits of Generic Recovery[7] gives a broad overview of failure recovery and how to make it transparent to a system user.

Additionally, there are a number of journaled filesystems other than Ext3 which may have different transaction methods. A non-exhaustive list includes XFS[16], which uses the FUA bit already; ReiserFS[12] and IBM’s Journaled File System (JFS)[2], which do not use the FUA bit; and the closed-source New Technology File System (NTFS)[11].

There is definitely room for more related studies, as data integrity is of utmost importance in a number of industries. Additionally, researchers in this field could also find new and better methods for guaranteeing data safety beyond TCQ and similar methods. Our own FUA method may turn out to have much better performance with some tuning than we managed to achieve.
Tagged Command Queuing is the standard method that Ext3 uses to guarantee that the disk journal is written before the data itself. Ext3 uses TCQ because it is efficient, mostly due to it being an asynchronous method that supports parallel operations. However, TCQ is not a required feature according to the SCSI-2 specification, and therefore Ext3 should provide a fallback in case that the disk being used does not support it. Ext3 does not, however, support any fallback methods to guarantee write ordering, and could, theoretically, on a disk not supporting TCQ, become inconsistent after a power loss or other sudden interruption.

Forcing the disk to write the journal synchronously, while allowing it to write normal data asynchronously, is a possible fallback to the TCQ method. SCSI-2 provides for a Force Unit Access bit in read and write requests. Setting this bit in a write request forces the disk to write the data to the physical medium before replying. The result is a guarantee that journal writes will hit the disk before the normal writes that they correspond to, but at a significant performance hit.

Our research showed that without any tuning of the filesystem, using the FUA bit on journal writes will decrease performance by about 43%. There is certainly room for improvement on our results, but even with such a significant performance hit, there are settings where data consistency takes precedence, and users should feel confident that their data will not be lost due to a corrupt filesystem. Future research can take this result and refine it to determine the best way to optimize such a filesystem. Perhaps by
placing a number of smaller journals around the disk one may prevent a synchronous
journal write from seeking to an extreme point on the platter.

Since reading speed is unaffected, scenarios with heavy read throughput but infre-
quenent writes could conceivably even use this technology to save money. If it would be
cheaper to purchase disks without TCQ, they would be a viable option once the fear of
corruption due to power loss is alleviated.

This work is mostly academic, though. Virtually every SCSI controller and disk
supports TCQ from the SCSI-2 specification[14]. There is no guarantee that SCSI-1
hardware is not still in circulation, although the statistics are likely approaching zero.

In conclusion, Ext3 is not as safe a journaling filesystem as it could be. It only
supports one method, TCQ, of write-ordering to guarantee consistency of its journal, but
that one method is an optional feature, according to the SCSI-2 specification. Virtually
all hardware supports it, but as it is optional, that is still not a safe implementation.
Forcing writes using the FUA bit is a solution. It takes longer to write to the disk, but
it is a feature that only needs to be used when TCQ is not available.
Bibliography


