XFS Delayed Logging Design

Introduction to Re-logging in XFS

XFS logging is a combination of logical and physical logging. Some objects, such as inodes and dquots, are logged in logical format where the details logged are made up of the changes to in-core structures rather than on-disk structures. Other objects - typically buffers - have their physical changes logged. The reason for these differences is to reduce the amount of log space required for objects that are frequently logged. Some parts of inodes are more frequently logged than others, and inodes are typically more frequently logged than any other object (except maybe the superblock buffer) so keeping the amount of metadata logged low is of prime importance.

The reason that this is such a concern is that XFS allows multiple separate modifications to a single object to be carried in the log at any given time. This allows the log to avoid needing to flush each change to disk before recording a new change to the object. XFS does this via a method called "re-logging". Conceptually, this is quite simple - all it requires is that any new change to the object is recorded with a *new copy* of all the existing changes in the new transaction that is written to the log.

That is, if we have a sequence of changes A through to F, and the object was written to disk after change D, we would see in the log the following series of transactions, their contents and the log sequence number (LSN) of the transaction:

Transaction	Contents	LSN
A	A	X
B	A+B	X+n
C	A+B+C	X+n+m
D	A+B+C+D	X+n+m+o
<object written to disk>		
E	E	Y (> X+n+m+o)
F	E+F	Y+p

In other words, each time an object is relogged, the new transaction contains the aggregation of all the previous changes currently held only in the log.

This relogging technique also allows objects to be moved forward in the log so that an object being relogged does not prevent the tail of the log from ever moving forward. This can be seen in the table above by the changing (increasing) LSN of each subsquent transaction - the LSN is effectively a direct encoding of the location in the log of the transaction.

This relogging is also used to implement long-running, multiple-commit transactions. These transaction are known as rolling transactions, and require a special log reservation known as a permanent transaction reservation. A typical example of a rolling transaction is the removal of extents from an inode which can only be done at a rate of two extents per transaction because of reservation size limitations. Hence a rolling extent removal transaction keeps relogging the inode and btree buffers as they get modified in each removal operation. This keeps them moving forward in the log as the operation progresses, ensuring that current operation never gets blocked by itself if the log wraps around.

Hence it can be seen that the relogging operation is fundamental to the correct working of the XFS journalling subsystem. From the above description, most people should be able to see why the XFS metadata operations writes so much to the log - repeated operations to the same objects write the same changes to the log over and over again. Worse is the fact that objects tend to get dirtier as they get relogged, so each subsequent transaction is writing more metadata into the log.

Another feature of the XFS transaction subsystem is that most transactions are asynchronous. That is, they don't commit to disk until either a log buffer is filled (a log buffer can hold multiple transactions) or a synchronous operation forces the log buffers holding the transactions to disk. This means that XFS is doing aggregation of transactions in memory - batching them, if you like - to minimise the impact of the log IO on transaction throughput.

The limitation on asynchronous transaction throughput is the number and size of log buffers made available by the log manager. By default there are 8 log buffers available and the size of each is 32kB - the size can be increased up to 256kB by use of a mount option.

Effectively, this gives us the maximum bound of outstanding metadata changes that can be made to the filesystem at any point in time - if all the log buffers are full and under IO, then no more transactions can be committed until the current batch completes. It is now common for a single current CPU core to be to able to issue enough transactions to keep the log buffers full and under IO permanently. Hence the XFS journalling subsystem can be considered to be IO bound.

Delayed Logging: Concepts

The key thing to note about the asynchronous logging combined with the relogging technique XFS uses is that we can be relogging changed objects multiple times before they are committed to disk in the log buffers. If we return to the previous relogging example, it is entirely possible that transactions A through D are committed to disk in the same log buffer.

That is, a single log buffer may contain multiple copies of the same object, but only one of those copies needs to be there - the last one "D", as it contains all the changes from the previous changes. In other words, we have one necessary copy in the log buffer, and three stale copies that are simply wasting space. When we are doing repeated operations on the same set of objects, these "stale objects" can be over 90% of the space used in the log buffers. It is clear that reducing the number of stale objects written to the log would greatly reduce the amount of metadata we write to the log, and this is the fundamental goal of delayed logging.

From a conceptual point of view, XFS is already doing relogging in memory (where memory == log buffer), only it is doing it extremely inefficiently. It is using logical to physical formatting to do the relogging because there is no infrastructure to keep track of logical changes in memory prior to physically formatting the changes in a transaction to the log buffer. Hence we cannot avoid accumulating stale objects in the log buffers.

Delayed logging is the name we've given to keeping and tracking transactional changes to objects in memory outside the log buffer infrastructure. Because of the relogging concept fundamental to the XFS journalling subsystem, this is actually relatively easy to do - all the changes to logged items are already tracked in the current infrastructure. The big problem is how to accumulate them and get them to the log in a consistent, recoverable manner. Describing the problems and how they have been solved is the focus of this document.

One of the key changes that delayed logging makes to the operation of the journalling subsystem is that it disassociates the amount of outstanding metadata changes from the size and number of log buffers available. In other words, instead of there only being a maximum of 2MB of transaction changes not written to the log at any point in time, there may be a much greater amount being accumulated in memory. Hence the potential for loss of metadata on a crash is much greater than for the existing logging mechanism.

It should be noted that this does not change the guarantee that log recovery will result in a consistent filesystem. What it does mean is that as far as the recovered filesystem is concerned, there may be many thousands of transactions that simply did not occur as a result of the crash. This makes it even more important that applications that care about their data use fsync() where they need to ensure application level data integrity is maintained.

It should be noted that delayed logging is not an innovative new concept that warrants rigorous proofs to determine whether it is correct or not. The method of accumulating changes in memory for some period before writing them to the log is used effectively in many filesystems including ext3 and ext4. Hence no time is spent in this document trying to convince the reader that the concept is sound. Instead it is simply considered a "solved problem" and as such implementing it in XFS is purely an exercise in software engineering.

The fundamental requirements for delayed logging in XFS are simple:

1. Reduce the amount of metadata written to the log by at least an order of magnitude.
2. Supply sufficient statistics to validate Requirement #1.
3. Supply sufficient new tracing infrastructure to be able to debug problems with the new code.
4. No on-disk format change (metadata or log format).
5. Enable and disable with a mount option.
6. No performance regressions for synchronous transaction workloads.

Delayed Logging: Design

Storing Changes

The problem with accumulating changes at a logical level (i.e. just using the existing log item dirty region tracking) is that when it comes to writing the changes to the log buffers, we need to ensure that the object we are formatting is not changing while we do this. This requires locking the object to prevent concurrent modification. Hence flushing the logical changes to the log would require us to lock every object, format them, and then unlock them again.

This introduces lots of scope for deadlocks with transactions that are already running. For example, a transaction has object A locked and modified, but needs the delayed logging tracking lock to commit the transaction. However, the flushing thread has the delayed logging tracking lock already held, and is trying to get the lock on object A to flush it to the log buffer. This appears to be an unsolvable deadlock condition, and it was solving this problem that was the barrier to implementing delayed logging for so long.

The solution is relatively simple - it just took a long time to recognise it. Put simply, the current logging code formats the changes to each item into an vector array that points to the changed regions in the item. The log write code simply copies the memory these vectors point to into the log buffer during transaction commit while the item is locked in the transaction. Instead of using the log buffer as the destination of the formatting code, we can use an allocated memory buffer big enough to fit the formatted vector.

If we then copy the vector into the memory buffer and rewrite the vector to point to the memory buffer rather than the object itself, we now have a copy of the changes in a format that is compatible with the log buffer writing code. that does not require us to lock the item to access. This formatting and rewriting can all be done while the object is locked during transaction commit, resulting in a vector that is transactionally consistent and can be accessed without needing to lock the owning item.

Hence we avoid the need to lock items when we need to flush outstanding asynchronous transactions to the log. The differences between the existing formatting method and the delayed logging formatting can be seen in the diagram below.

Current format log vector:

```
Object    +---------------------------------------------+
Vector 1      +----+
Vector 2                    +----+
Vector 3                                   +----------+
```

After formatting:

```
Log Buffer    +-V1-+-V2-+----V3----+
```

Delayed logging vector:

```
Object    +---------------------------------------------+
Vector 1      +----+
Vector 2                    +----+
Vector 3                                   +----------+
```

After formatting:

```
Memory Buffer +-V1-+-V2-+----V3----+
Vector 1      +----+
Vector 2           +----+
Vector 3                +----------+
```

The memory buffer and associated vector need to be passed as a single object, but still need to be associated with the parent object so if the object is relogged we can replace the current memory buffer with a new memory buffer that contains the latest changes.

The reason for keeping the vector around after we've formatted the memory buffer is to support splitting vectors across log buffer boundaries correctly. If we don't keep the vector around, we do not know where the region boundaries are in the item, so we'd need a new encapsulation method for regions in the log buffer writing (i.e. double encapsulation). This would be an on-disk format change and as such is not desirable. It also means we'd have to write the log region headers in the formatting stage, which is problematic as there is per region state that needs to be placed into the headers during the log write.

Hence we need to keep the vector, but by attaching the memory buffer to it and rewriting the vector addresses to point at the memory buffer we end up with a self-describing object that can be passed to the log buffer write code to be handled in exactly the same manner as the existing log vectors are handled. Hence we avoid needing a new on-disk format to handle items that have been relogged in memory.

Tracking Changes

Now that we can record transactional changes in memory in a form that allows them to be used without limitations, we need to be able to track and accumulate them so that they can be written to the log at some later point in time. The log item is the natural place to store this vector and buffer, and also makes sense

to be the object that is used to track committed objects as it will always exist once the object has been included in a transaction.

The log item is already used to track the log items that have been written to the log but not yet written to disk. Such log items are considered "active" and as such are stored in the Active Item List (AIL) which is a LSN-ordered double linked list. Items are inserted into this list during log buffer IO completion, after which they are unpinned and can be written to disk. An object that is in the AIL can be relogged, which causes the object to be pinned again and then moved forward in the AIL when the log buffer IO completes for that transaction.

Essentially, this shows that an item that is in the AIL can still be modified and relogged, so any tracking must be separate to the AIL infrastructure. As such, we cannot reuse the AIL list pointers for tracking committed items, nor can we store state in any field that is protected by the AIL lock. Hence the committed item tracking needs it's own locks, lists and state fields in the log item.

Similar to the AIL, tracking of committed items is done through a new list called the Committed Item List (CIL). The list tracks log items that have been committed and have formatted memory buffers attached to them. It tracks objects in transaction commit order, so when an object is relogged it is removed from it's place in the list and re-inserted at the tail. This is entirely arbitrary and done to make it easy for debugging - the last items in the list are the ones that are most recently modified. Ordering of the CIL is not necessary for transactional integrity (as discussed in the next section) so the ordering is done for convenience/sanity of the developers.

Delayed Logging: Checkpoints

When we have a log synchronisation event, commonly known as a "log force", all the items in the CIL must be written into the log via the log buffers. We need to write these items in the order that they exist in the CIL, and they need to be written as an atomic transaction. The need for all the objects to be written as an atomic transaction comes from the requirements of relogging and log replay - all the changes in all the objects in a given transaction must either be completely replayed during log recovery, or not replayed at all. If a transaction is not replayed because it is not complete in the log, then no later transactions should be replayed, either.

To fulfill this requirement, we need to write the entire CIL in a single log transaction. Fortunately, the XFS log code has no fixed limit on the size of a transaction, nor does the log replay code. The only fundamental limit is that the transaction cannot be larger than just under half the size of the log. The reason for this limit is that to find the head and tail of the log, there must be at least one complete transaction in the log at any given time. If a transaction is larger than half the log, then there is the possibility that a crash during the write of a such a transaction could partially overwrite the only complete previous transaction in the log. This will result in a recovery failure and an inconsistent filesystem and hence we must enforce the maximum size of a checkpoint to be slightly less than a half the log.

Apart from this size requirement, a checkpoint transaction looks no different to any other transaction - it contains a transaction header, a series of formatted log items and a commit record at the tail. From a recovery perspective, the checkpoint transaction is also no different - just a lot bigger with a lot more items in it. The worst case effect of this is that we might need to tune the recovery transaction object hash size.

Because the checkpoint is just another transaction and all the changes to log items are stored as log vectors, we can use the existing log buffer writing code to write the changes into the log. To do this efficiently, we need to minimise the time we hold the CIL locked while writing the checkpoint transaction. The current log write code enables us to do this easily with the way it separates the writing of the transaction contents (the log vectors) from the transaction commit record, but tracking this requires us to have a per-checkpoint context that travels through the log write process through to checkpoint completion.

Hence a checkpoint has a context that tracks the state of the current checkpoint from initiation to checkpoint completion. A new context is initiated at the same time a checkpoint transaction is started. That is, when we remove all the current items from the CIL during a checkpoint operation, we move all those changes into the current checkpoint context. We then initialise a new context and attach that to the CIL for aggregation of new transactions.

This allows us to unlock the CIL immediately after transfer of all the committed items and effectively allow new transactions to be issued while we are formatting the checkpoint into the log. It also allows concurrent checkpoints to be written into the log buffers in the case of log force heavy workloads, just like the existing transaction commit code does. This, however, requires that we strictly order the commit records in the log so that checkpoint sequence order is maintained during log replay.

To ensure that we can be writing an item into a checkpoint transaction at the same time another transaction modifies the item and inserts the log item into the new CIL, then checkpoint transaction commit code cannot use log items to store the list of log vectors that need to be written into the transaction. Hence log vectors need to be able to be chained together to allow them to be detatched from the log items. That is, when the CIL is flushed the memory buffer and log vector attached to each log item needs to be attached to the checkpoint context so that the log item can be released. In diagrammatic form, the CIL would look like this before the flush:

```
        CIL Head
           |
           V
        Log Item <-> log vector 1       -> memory buffer
           |                            -> vector array
           V
        Log Item <-> log vector 2       -> memory buffer
           |                            -> vector array
           V
        ......
           |
           V
        Log Item <-> log vector N-1     -> memory buffer
           |                            -> vector array
           V
        Log Item <-> log vector N       -> memory buffer
                                        -> vector array
```

And after the flush the CIL head is empty, and the checkpoint context log vector list would look like:

```
        Checkpoint Context
           |
           V
        log vector 1    -> memory buffer
           |            -> vector array
           |            -> Log Item
           V
        log vector 2    -> memory buffer
           |            -> vector array
           |            -> Log Item
           V
        ......
           |
           V
        log vector N-1  -> memory buffer
           |            -> vector array
           |            -> Log Item
           V
        log vector N    -> memory buffer
                        -> vector array
                        -> Log Item
```

Once this transfer is done, the CIL can be unlocked and new transactions can start, while the checkpoint flush code works over the log vector chain to commit the checkpoint.

Once the checkpoint is written into the log buffers, the checkpoint context is attached to the log buffer that the commit record was written to along with a completion callback. Log IO completion will call that callback, which can then run transaction committed processing for the log items (i.e. insert into AIL and unpin) in the log vector chain and then free the log vector chain and checkpoint context.

Discussion Point: I am uncertain as to whether the log item is the most efficient way to track vectors, even though it seems like the natural way to do it. The fact that we walk the log items (in the CIL) just to chain the log vectors and break the link between the log item and the log vector means that we take a cache line hit for the log item list modification, then another for the log vector chaining. If we track by the log vectors, then we only need to break the link between the log item and the log vector, which means we should dirty only the log item cachelines. Normally I wouldn't be concerned about one vs two dirty cachelines except for the fact I've seen upwards of 80,000 log vectors in one checkpoint transaction. I'd guess this is a "measure and compare" situation that can be done after a working and reviewed implementation is in the dev tree....

Delayed Logging: Checkpoint Sequencing

One of the key aspects of the XFS transaction subsystem is that it tags committed transactions with the log sequence number of the transaction commit. This allows transactions to be issued asynchronously even though there may be future operations that cannot be completed until that transaction is fully committed to the log. In the rare case that a dependent operation occurs (e.g. re-using a freed metadata extent for a data extent), a special, optimised log force can be issued to force the dependent transaction to disk immediately.

To do this, transactions need to record the LSN of the commit record of the transaction. This LSN comes directly from the log buffer the transaction is written into. While this works just fine for the existing transaction mechanism, it does not work for delayed logging because transactions are not written directly into the log buffers. Hence some other method of sequencing transactions is required.

As discussed in the checkpoint section, delayed logging uses per-checkpoint contexts, and as such it is simple to assign a sequence number to each checkpoint. Because the switching of checkpoint contexts must be done atomically, it is simple to ensure that each new context has a monotonically increasing sequence number assigned to it without the need for an external atomic counter - we can just take the current context sequence number and add one to it for the new context.

Then, instead of assigning a log buffer LSN to the transaction commit LSN during the commit, we can assign the current checkpoint sequence. This allows operations that track transactions that have not yet completed know what checkpoint sequence needs to be committed before they can continue. As a result, the code that forces the log to a specific LSN now needs to ensure that the log forces to a specific checkpoint.

To ensure that we can do this, we need to track all the checkpoint contexts that are currently committing to the log. When we flush a checkpoint, the context gets added to a "committing" list which can be searched. When a checkpoint commit completes, it is removed from the committing list. Because the checkpoint context records the LSN of the commit record for the checkpoint, we can also wait on the log buffer that contains the commit record, thereby using the existing log force mechanisms to execute synchronous forces.

It should be noted that the synchronous forces may need to be extended with mitigation algorithms similar to the current log buffer code to allow aggregation of multiple synchronous transactions if there are already synchronous transactions being flushed. Investigation of the performance of the current design is needed before making any decisions here.

The main concern with log forces is to ensure that all the previous checkpoints are also committed to disk before the one we need to wait for. Therefore we need to check that all the prior contexts in the committing list are also complete before waiting on the one we need to complete. We do this synchronisation in the log force code so that we don't need to wait anywhere else for such serialisation - it only matters when we do a log force.

The only remaining complexity is that a log force now also has to handle the case where the forcing sequence number is the same as the current context. That is, we need to flush the CIL and potentially wait for it to complete. This is a simple addition to the existing log forcing code to check the sequence numbers and push if required. Indeed, placing the current sequence checkpoint flush in the log force code enables the current mechanism for issuing synchronous transactions to remain untouched (i.e. commit an asynchronous transaction, then force the log at the LSN of that transaction) and so the higher level code behaves the same regardless of whether delayed logging is being used or not.

Delayed Logging: Checkpoint Log Space Accounting

The big issue for a checkpoint transaction is the log space reservation for the transaction. We don't know how big a checkpoint transaction is going to be ahead of time, nor how many log buffers it will take to write out, nor the number of split log vector regions are going to be used. We can track the amount of log space required as we add items to the commit item list, but we still need to reserve the space in the log for the checkpoint.

A typical transaction reserves enough space in the log for the worst case space usage of the transaction. The reservation accounts for log record headers, transaction and region headers, headers for split regions, buffer tail padding, etc. as well as the actual space for all the changed metadata in the transaction. While some of this is fixed overhead, much of it is dependent on the size of the transaction and the number of regions being logged (the number of log vectors in the transaction).

An example of the differences would be logging directory changes versus logging inode changes. If you modify lots of inode cores (e.g. chmod -R g+w *), then there are lots of transactions that only contain an inode core and an inode log format structure. That is, two vectors totaling roughly 150 bytes. If we modify 10,000 inodes, we have about 1.5MB of metadata to write in 20,000 vectors. Each vector is 12 bytes, so the total to be logged is approximately 1.75MB. In comparison, if we are logging full directory buffers, they are typically 4KB each, so we in 1.5MB of directory buffers we'd have roughly 400 buffers and a buffer format structure for each buffer - roughly 800 vectors or 1.51MB total space. From this, it should be obvious that a static log space reservation is not particularly flexible and is difficult to select the "optimal value" for all workloads.

Further, if we are going to use a static reservation, which bit of the entire reservation does it cover? We account for space used by the transaction reservation by tracking the space currently used by the object in the CIL and then calculating the increase or decrease in space used as the object is relogged. This allows for a checkpoint reservation to only have to account for log buffer metadata used such as log header records.

However, even using a static reservation for just the log metadata is problematic. Typically log record headers use at least 16KB of log space per 1MB of log space consumed (512 bytes per 32k) and the reservation needs to be large enough to handle arbitrary sized checkpoint transactions. This reservation needs to be made before the checkpoint is started, and we need to be able to reserve the space without sleeping. For a 8MB checkpoint, we need a reservation of around 150KB, which is a non-trivial amount of space.

A static reservation needs to manipulate the log grant counters - we can take a permanent reservation on the space, but we still need to make sure we refresh the write reservation (the actual space available to the transaction) after every checkpoint transaction completion. Unfortunately, if this space is not available when required, then the regrant code will sleep waiting for it.

The problem with this is that it can lead to deadlocks as we may need to commit checkpoints to be able to free up log space (refer back to the description of rolling transactions for an example of this). Hence we *must* always have space available in the log if we are to use static reservations, and that is very difficult and complex to arrange. It is possible to do, but there is a simpler way.

The simpler way of doing this is tracking the entire log space used by the items in the CIL and using this to dynamically calculate the amount of log space required by the log metadata. If this log metadata space changes as a result of a transaction commit inserting a new memory buffer into the CIL, then the difference in space required is removed from the transaction that causes the change. Transactions at this level will *always* have enough space available in their reservation for this as they have already reserved the maximal amount of log metadata space they require, and such a delta reservation will always be less than or equal to the maximal amount in the reservation.

Hence we can grow the checkpoint transaction reservation dynamically as items are added to the CIL and avoid the need for reserving and regranting log space up front. This avoids deadlocks and removes a blocking point from the checkpoint flush code.

As mentioned early, transactions can't grow to more than half the size of the log. Hence as part of the reservation growing, we need to also check the size of the reservation against the maximum allowed transaction size. If we reach the maximum threshold, we need to push the CIL to the log. This is effectively a "background flush" and is done on demand. This is identical to
a CIL push triggered by a log force, only that there is no waiting for the checkpoint commit to complete. This background push is checked and executed by transaction commit code.

If the transaction subsystem goes idle while we still have items in the CIL, they will be flushed by the periodic log force issued by the xfssyncd. This log force will push the CIL to disk, and if the transaction subsystem stays idle, allow the idle log to be covered (effectively marked clean) in exactly the same manner that is done for the existing logging method. A discussion point is whether this log force needs to be done more frequently than the current rate which is once every 30s.

Delayed Logging: Log Item Pinning

Currently log items are pinned during transaction commit while the items are still locked. This happens just after the items are formatted, though it could be done any time before the items are unlocked. The result of this mechanism is that items get pinned once for every transaction that is committed to the log buffers. Hence items that are relogged in the log buffers will have a pin count for every outstanding transaction they were dirtied in. When each of these transactions is completed, they will unpin the item once. As a result, the item only becomes unpinned when all the transactions complete and there are no pending transactions. Thus the pinning and unpinning of a log item is symmetric as there is a 1:1 relationship with transaction commit and log item completion.

For delayed logging, however, we have an assymetric transaction commit to completion relationship. Every time an object is relogged in the CIL it goes through the commit process without a corresponding completion being registered. That is, we now have a many-to-one relationship between transaction commit and log item completion. The result of this is that pinning and unpinning of the log items becomes unbalanced if we retain the "pin on transaction commit, unpin on transaction completion" model.

To keep pin/unpin symmetry, the algorithm needs to change to a "pin on insertion into the CIL, unpin on checkpoint completion". In other words, the pinning and unpinning becomes symmetric around a checkpoint context. We have to pin the object the first time it is inserted into the CIL - if it is already in the CIL during a transaction commit, then we do not pin it again. Because there can be multiple outstanding checkpoint contexts, we can still see elevated pin counts, but as each checkpoint completes the pin count will retain the correct value according to it's context.

Just to make matters more slightly more complex, this checkpoint level context for the pin count means that the pinning of an item must take place under the CIL commit/flush lock. If we pin the object outside this lock, we cannot guarantee which context the pin count is associated with. This is because of the fact pinning the item is dependent on whether the item is present in the current CIL or not. If we don't pin the CIL first before we check and pin the object, we have a race with CIL being flushed between the check and the pin (or not pinning, as the case may be). Hence we must hold the CIL flush/commit lock to guarantee that we pin the items correctly.

Delayed Logging: Concurrent Scalability

A fundamental requirement for the CIL is that accesses through transaction commits must scale to many concurrent commits. The current transaction commit code does not break down even when there are transactions coming from 2048 processors at once. The current transaction code does not go any faster than if there was only one CPU using it, but it does not slow down either.

As a result, the delayed logging transaction commit code needs to be designed for concurrency from the ground up. It is obvious that there are serialisation points in the design - the three important ones are:

1. Locking out new transaction commits while flushing the CIL
2. Adding items to the CIL and updating item space accounting
3. Checkpoint commit ordering

Looking at the transaction commit and CIL flushing interactions, it is clear that we have a many-to-one interaction here. That is, the only restriction on the number of concurrent transactions that can be trying to commit at once is the amount of space available in the log for their reservations. The practical limit here is in the order of several hundred concurrent transactions for a 128MB log, which means that it is generally one per CPU in a machine.

The amount of time a transaction commit needs to hold out a flush is a relatively long period of time - the pinning of log items needs to be done while we are holding out a CIL flush, so at the moment that means it is held across the formatting of the objects into memory buffers (i.e. while memcpy()s are in progress). Ultimately a two pass algorithm where the formatting is done separately to the pinning of objects could be used to reduce the hold time of the transaction commit side.

Because of the number of potential transaction commit side holders, the lock really needs to be a sleeping lock - if the CIL flush takes the lock, we do not want every other CPU in the machine spinning on the CIL lock. Given that flushing the CIL could involve walking a list of tens of thousands of log items, it will get held for a significant time and so spin contention is a significant concern. Preventing lots of CPUs spinning doing nothing is the main reason for choosing a sleeping lock even though nothing in either the transaction commit or CIL flush side sleeps with the lock held.

It should also be noted that CIL flushing is also a relatively rare operation compared to transaction commit for asynchronous transaction workloads - only time will tell if using a read-write semaphore for exclusion will limit transaction commit concurrency due to cache line bouncing of the lock on the read side.

The second serialisation point is on the transaction commit side where items are inserted into the CIL. Because transactions can enter this code concurrently, the CIL needs to be protected separately from the above commit/flush exclusion. It also needs to be an exclusive lock but it is only held for a very short time and so a spin lock is appropriate here. It is possible that this lock will become a contention point, but given the short hold time once per transaction I think that contention is unlikely.

The final serialisation point is the checkpoint commit record ordering code that is run as part of the checkpoint commit and log force sequencing. The code path that triggers a CIL flush (i.e. whatever triggers the log force) will enter an ordering loop after writing all the log vectors into the log buffers but before writing the commit record. This loop walks the list of committing checkpoints and needs to block waiting for checkpoints to complete their commit record write. As a result it needs a lock and a wait variable. Log force sequencing also requires the same lock, list walk, and blocking mechanism to ensure completion of checkpoints.

These two sequencing operations can use the mechanism even though the events they are waiting for are different. The checkpoint commit record sequencing needs to wait until checkpoint contexts contain a commit LSN (obtained through completion of a commit record write) while log force sequencing needs to wait until previous checkpoint contexts are removed from the committing list (i.e. they’ve completed). A simple wait variable and broadcast wakeups (thundering herds) has been used to implement these two serialisation queues. They use the same lock as the CIL, too. If we see too much contention on the CIL lock, or too many context switches as a result of the broadcast wakeups these operations can be put under a new spinlock and given separate wait lists to reduce lock contention and the number of processes woken by the wrong event.

Lifecycle Changes

The existing log item life cycle is as follows:

```
        1. Transaction allocate
        2. Transaction reserve
        3. Lock item
        4. Join item to transaction
                If not already attached,
                        Allocate log item
                        Attach log item to owner item
                Attach log item to transaction
        5. Modify item
                Record modifications in log item
        6. Transaction commit
                Pin item in memory
                Format item into log buffer
                Write commit LSN into transaction
                Unlock item
                Attach transaction to log buffer

        <log buffer IO dispatched>
        <log buffer IO completes>

        7. Transaction completion
                Mark log item committed
                Insert log item into AIL
                        Write commit LSN into log item
                Unpin log item
        8. AIL traversal
                Lock item
                Mark log item clean
                Flush item to disk

        <item IO completion>

        9. Log item removed from AIL
                Moves log tail
                Item unlocked
```

Essentially, steps 1-6 operate independently from step 7, which is also independent of steps 8-9. An item can be locked in steps 1-6 or steps 8-9 at the same time step 7 is occurring, but only steps 1-6 or 8-9 can occur at the same time. If the log item is in the AIL or between steps 6 and 7 and steps 1-6 are re-entered, then the item is relogged. Only when steps 8-9 are entered and completed is the object considered clean.

With delayed logging, there are new steps inserted into the life cycle:

```
	1. Transaction allocate
	2. Transaction reserve
	3. Lock item
	4. Join item to transaction
		If not already attached,
			Allocate log item
			Attach log item to owner item
		Attach log item to transaction
	5. Modify item
		Record modifications in log item
	6. Transaction commit
		Pin item in memory if not pinned in CIL
		Format item into log vector + buffer
		Attach log vector and buffer to log item
		Insert log item into CIL
		Write CIL context sequence into transaction
		Unlock item

	<next log force>

	7. CIL push
		lock CIL flush
		Chain log vectors and buffers together
		Remove items from CIL
		unlock CIL flush
		write log vectors into log
		sequence commit records
		attach checkpoint context to log buffer

	<log buffer IO dispatched>
	<log buffer IO completes>

	8. Checkpoint completion
		Mark log item committed
		Insert item into AIL
			Write commit LSN into log item
		Unpin log item
	9. AIL traversal
		Lock item
		Mark log item clean
		Flush item to disk
	<item IO completion>
	10. Log item removed from AIL
		Moves log tail
		Item unlocked
```

From this, it can be seen that the only life cycle differences between the two logging methods are in the middle of the life cycle - they still have the same beginning and end and execution constraints. The only differences are in the commiting of the log items to the log itself and the completion processing.

Hence delayed logging should not introduce any constraints on log item behaviour, allocation or freeing that don't already exist.

As a result of this zero-impact "insertion" of delayed logging infrastructure and the design of the internal structures to avoid on disk format changes, we can basically switch between delayed logging and the existing mechanism with a mount option. Fundamentally, there is no reason why the log manager would not be able to swap methods automatically and transparently depending on load characteristics, but this should not be necessary if delayed logging works as designed.

Roadmap:

```
2.6.37 Remove experimental tag from mount option
        => should be roughly 6 months after initial merge
        => enough time to:
                => gain confidence and fix problems reported by early
                   adopters (a.k.a. guinea pigs)
                => address worst performance regressions and undesired
                   behaviours
                => start tuning/optimising code for parallelism
                => start tuning/optimising algorithms consuming
                   excessive CPU time

2.6.39 Switch default mount option to use delayed logging
        => should be roughly 12 months after initial merge
        => enough time to shake out remaining problems before next round of
           enterprise distro kernel rebases
```