Bluffs

BSD Logging Updated Fast File System

Stephan Uphoff

ups@{freebsd.org|yahoo-inc.com}

http://people.freebsd.org/~ups/pubs/asiabsdcon2007/

Bluffs Main Features

- Journaling File System
  - Fast restart after system failure as no file system checker (fsck) is needed for recovery
- Mostly compatible with FFS
  - Allows easy bidirectional transitioning of existing file system
  - Existing infrastructure can be reused for booting and emergency file system repair.

Consistency problems in on disk file systems

File system operations frequently need to modify multiple locations (sectors) on disk

Example: Creating a file on ffs/bluffs modifies

- Disk location of directory inode
- Disk location of directory block
- Disk location of inode for new file
- Cylinder group block locations of new inode
- …

But disks only support atomic writes of a single sector at a time!

A system or power failure during the disk modification process leave the file system partially modified and inconsistent.

Stable and Ordered Disk Writes

To limit the class of inconsistencies that can occur after a system failure, file systems order some write operations (sector modifications) to the disk.

Two disk writes are ordered if the first write is required to be stable before the second write is issued.

A write to disk is stable if the file system knows that the disk sector modifications will survive a system or power failure.

For disks with no or none volatile write cache (most SCSI drives) any write operation completed by the disk subsystem is assumed to be stable.

Disk with volatile write cache (most IDE drives) need an additional operation to flush the cache before writes are stable.

Strategies for enabling file systems recovery

FFS (without soft updates)

Uses sector write atomicity by not crossing sector boundaries on:

- Directory entries
- Inodes
- Indirect block pointers

Uses ordered writes to limit file system inconsistencies to cases that can successfully repaired by the file system checker (fsck)

Strategies for enabling file systems recovery

FFS with soft updates

Same use of sector write atomicity as “classic” FFS.

Uses ordered writes and repeated writes using initially only a subset of the final modifications to limit file system inconsistencies to cases that can successfully repaired by a file system checker in the background while the file system is mounted (background fsck)

In general inconsistencies are restricted to free fragments and inodes not being in the relevant bitmaps.

Strategies for enabling file systems recovery

Bluffs:

Uses simple disk sector write atomicity and ordered writes as building blocks to allow the recovery process to guarantee higher level multi sector atomicity.

After recovery either non or all of the modifications of disk locations that transition a file system from one consistent state to the next are applied.

Write Ahead Logging (WAL)

WAL is the technique used by bluffs to guarantee atomicity of a set of changes to multiple disk locations. This set of changes is also called completed transactions.

Intend records that describe these changes are written to the on disk log.

Only after all intend records of a transaction are on stable storage(can be read after a system crash) the set of changes can be applied to the disk locations.

A system failure after all intend records of a transaction are stable will guarantee that changes are applied to the on disk file system on recovery.

A system failure before all intend records are stable will prevent any changes to be applied to the file system.

Example: Creating a file with WAL

Step 1:

Write intend records that **describe** intended changes of the:

- Disk location of directory inode
- Disk location of directory block
- Disk location of inode for new file
- Cylinder group block locations of new inode
- …

to the log

Step 2:

Wait until all of the intend records are on stable storage.

Step 3:

Apply the changes to

- Disk location of directory inode
- Disk location of directory block
- Disk location of inode for new file
- Cylinder group block locations of new inode
- …

Transaction Implementation

Once a transaction is completed it acquires an exclusive log lock and sequentially writes all its intend records to log buffers.

The last intend record of a transaction is marked with an end of transaction flag.

Bluffs implements lazy transactions and the log buffers are not automatically flushed to the log.

Since all intend records of a transaction are adjacent in the log all transactions are ordered and it is easy to detect the last stable transaction.

A transaction is stable once all of its intend records are stable in the on disk log.

Intend Record

The intend records written by Bluffs take the form of setting specific data in disk sectors to a value described in the record.

They take the form of clearing or setting bit ranges in a specific sector or copying data contained in the record to locations in the sector.

The operations described by the intend records are idempotent.

They can be repeatedly applied to the same sector without changing the results.

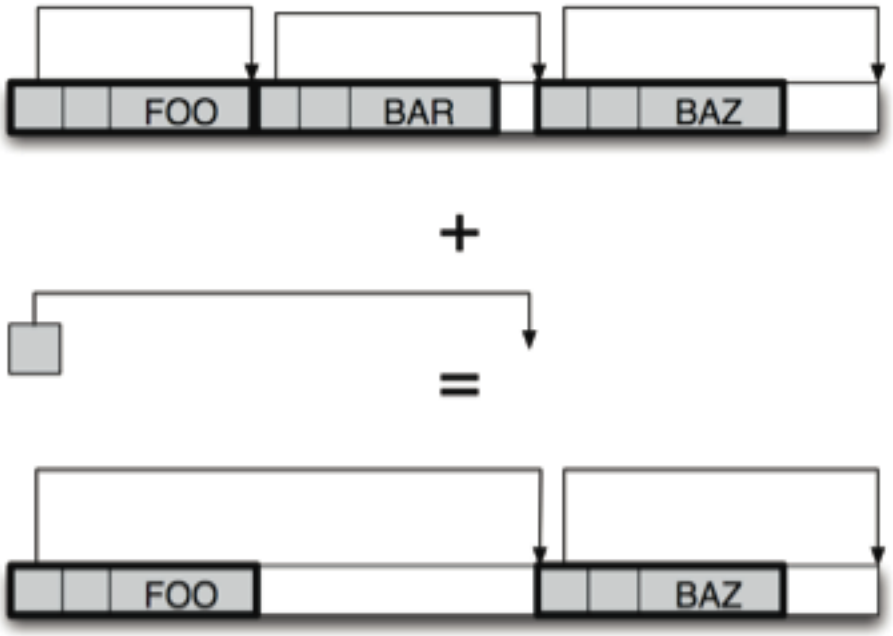
Examples of idempotent operations:

X = 3; X = X & 1; X = X | 1;

Examples of non idempotent operations:

X = X + 1; X = ~X; X = X ^ 1;

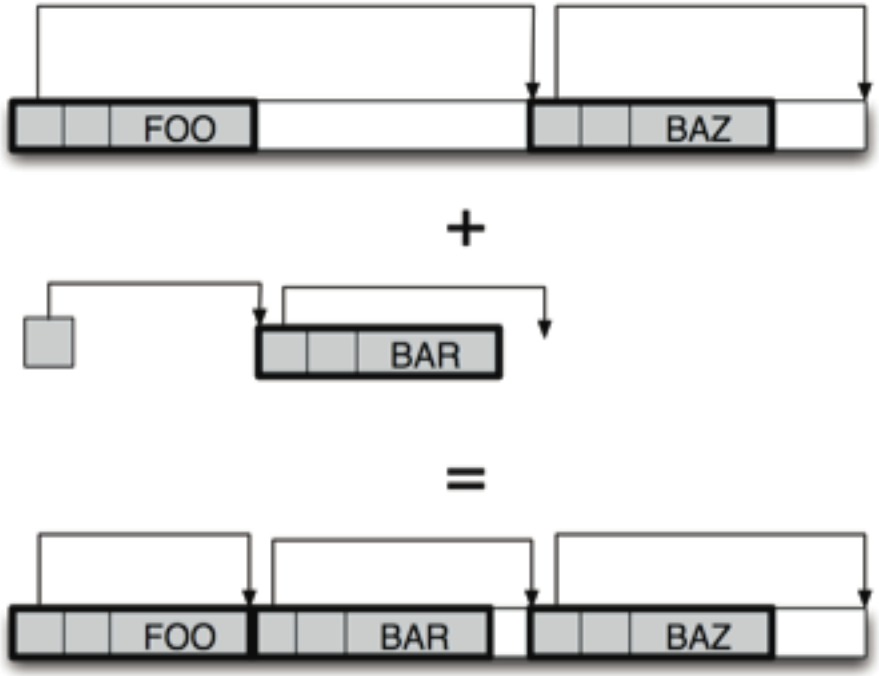
Intend record for removing a directory entry



The common case of removing a directory entry only requires modification of a the length field of the previous directory entry structure.

The intend record simply contains the disk sector ID and the offset and new value of the field.

Intend record for adding a directory entry



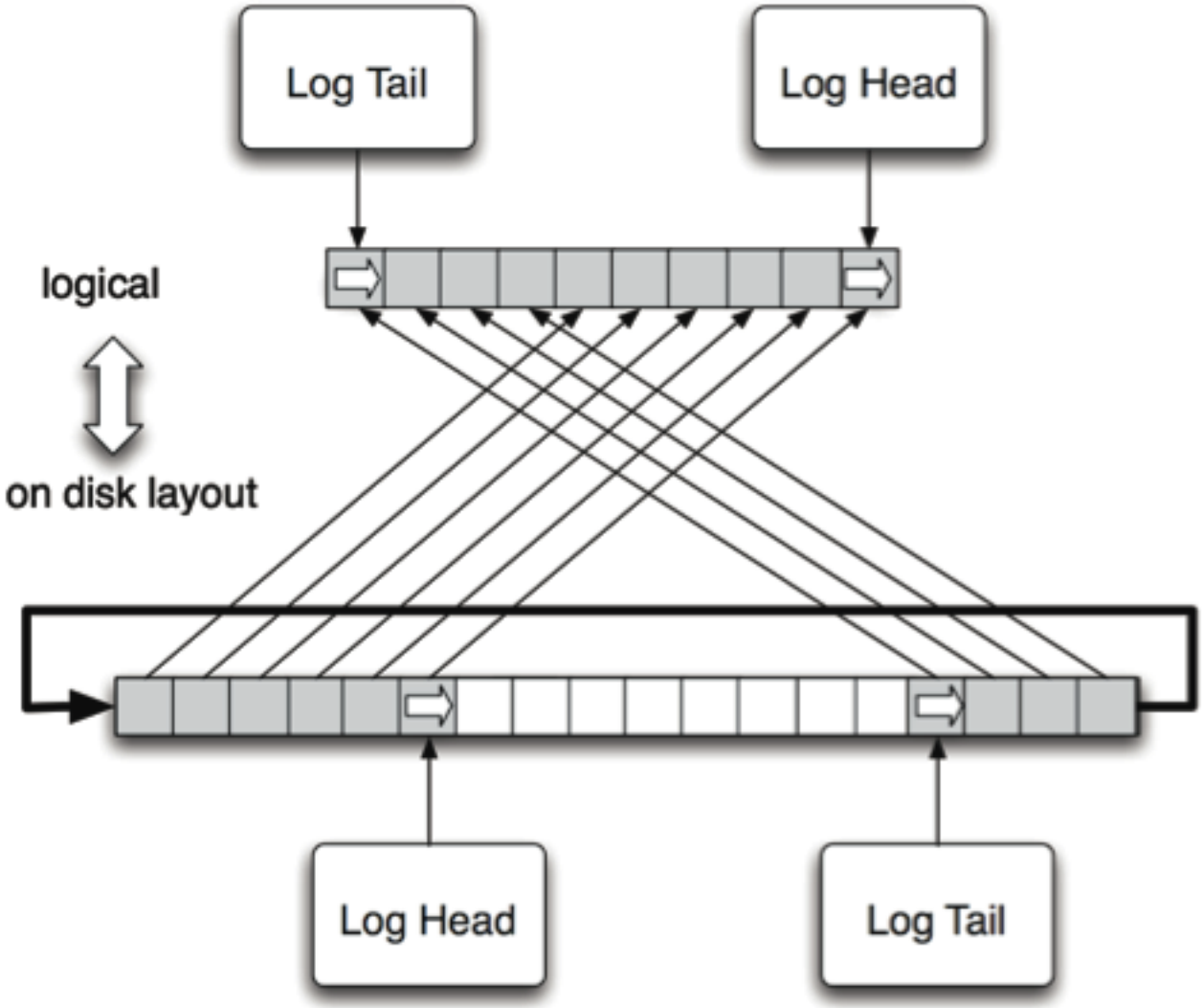
The common case of adding a directory entry only requires modification of a the length field of the previous directory entry structure and the copying of the new directory entry.

The intend record simply contains the disk sector ID and the offsets and new values of the changed byte locations.

Other examples of intend records

- Setting block pointers in indirect blocks -> intend record contain the disk sector ID and offset and new value of the pointer.
- Allocating or freeing inodes or fragments. -> intend record just contains the information since location of the relevant bitmaps are known. Cylinder summary information is updated as a side effect of applying the intended change.
- Changing inode fields (uid, gid,(m|c|a)times, link count, block pointers ..) -> intend record contains inode number and offsets and new values
- Changing number of directories in a cyclinder group -> Intend record contains cyclinder group number and the new value.

Bluffs uses an internal fixes size circular log



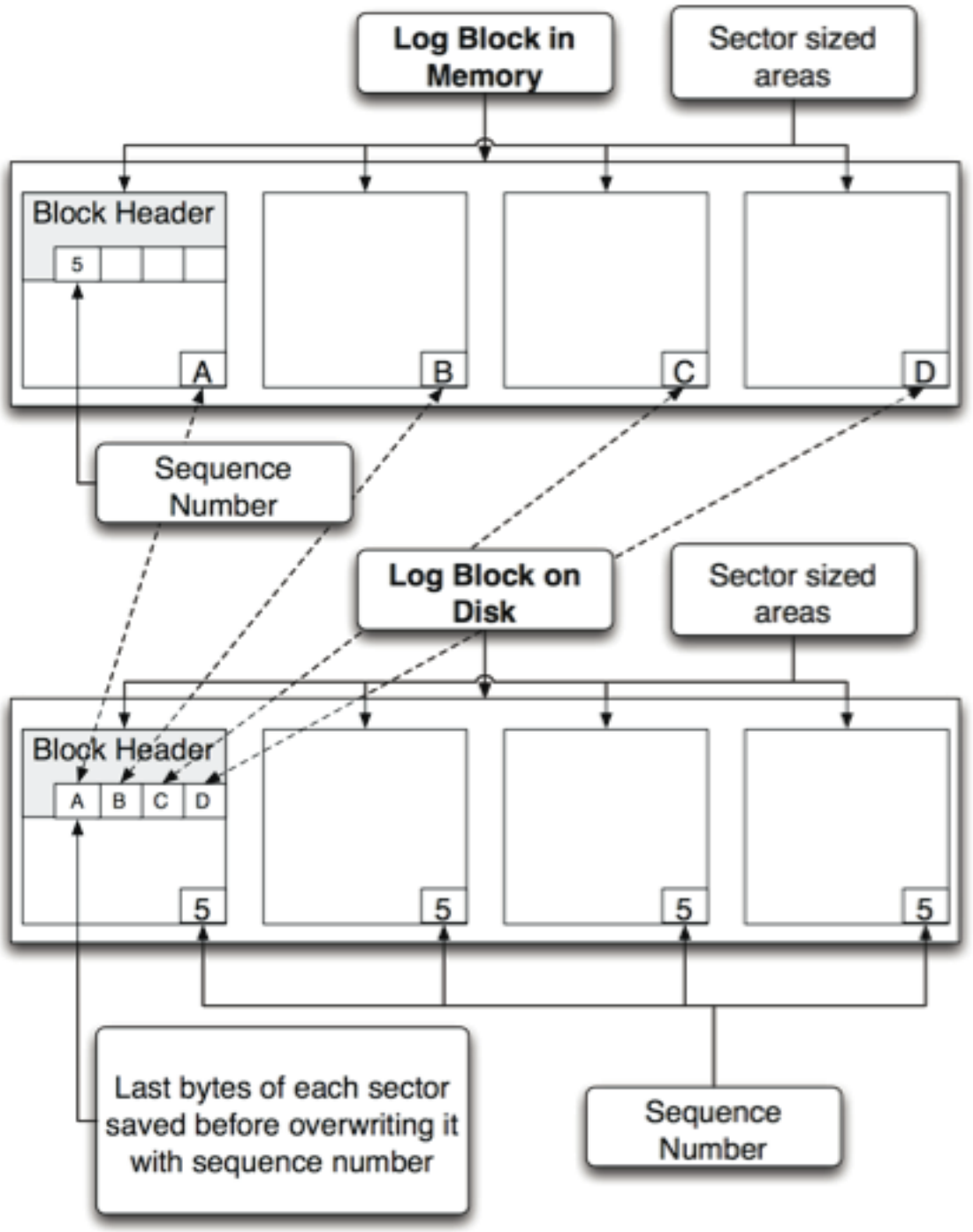
Atomicity of log block writes

Log Blocks that contain valid parts of the log are never overwritten.

As we don't care to preserve invalid data we can view the following as two atomic states that we need to detect after a system failure:

- All sectors of the block are updated and contain the new log block data
- None or not all sectors of the blocks have been updated

Bluffs marks each sector of a log block with a one byte sequence number to detect if a block has been updated. Whenever Bluffs wraps around the on disk log area it increments this sequence number. A block is invalid if any sector contains an old sequence number.



Log Block transformation

Log Blocks have a slightly different on disk and in memory layout.

The in memory format contains a header followed by data.

The on disk format additionally marks each sector with a sequence number and contains saved data in the header.

Checkpoint Record

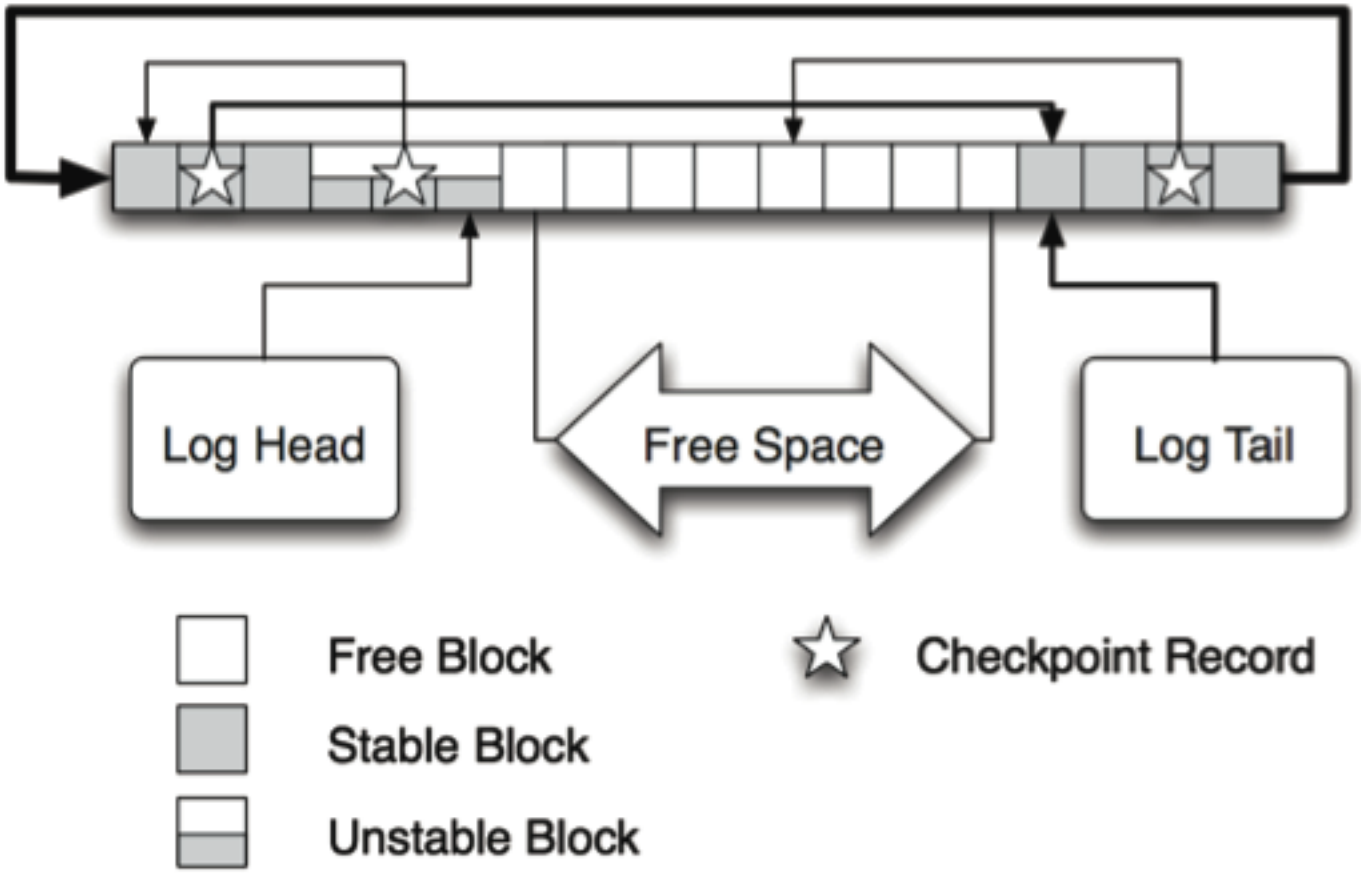
Bluffs uses checkpoint records to indicate the oldest intend record needed for recovery. The newest stable checkpoint record describes the log tail.

Moving the log tail to the right frees up space and is called log truncation.

Since Bluffs circular log uses that space to add new records to the log at the log head this is a required operation.

The log is truncate by making the oldest intend records obsolete by flushing changes to disk followed by writing a checkpoint record.

The most recent stable checkpoint record determines the log tail.



Additional Checkpoint Information

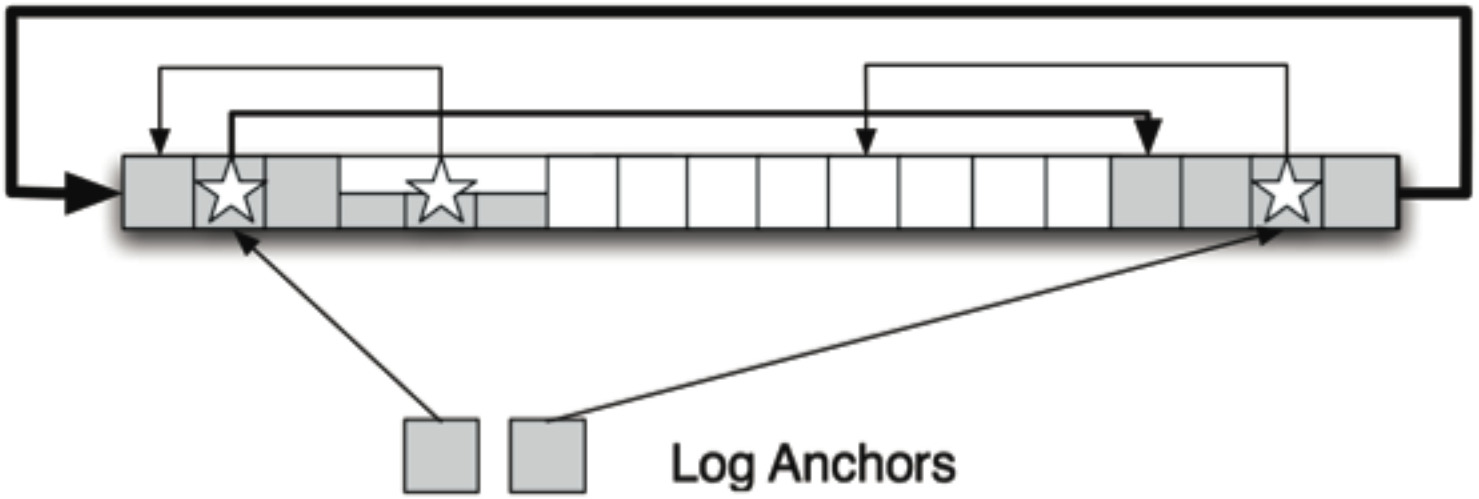
- The checkpoint actually contains a list of all sectors that need to be recovered.
- It also describes the oldest intend record needed for recovering those sectors on a per sector base.

  This drastically reduces the workload needed for recovering a file system.

Log Anchors

For recovery bluffs needs to find the most recent stable Checkpoint record. Theoretically the location can be recovered on startup by scanning the whole circular log.

However to accelerate restart a pair of ``Log anchors'' are used. The log anchors have the property that the last updated stable log anchor will point to the location of a stable checkpoint record.



Log Anchors (Cont)

Unfortunately this adds another write dependency since the log anchors can only be written after a checkpoint record is stable.

Because of this Bluffs may restrict the usage of ``Log Anchors'' to cleanly unmounted file systems or allow configurable behavior in the future.

Complex, restartable Operations

Bluffs tries to minimize the amount of work done in a single transaction. Complex operations are broken down into simple transactions that are applied sequentially.

In order to allow complex operations to be atomic from a file system semantic standpoint they need to be restarted after a system failure.

A fixed size on disk array is read on startup and each slot may contain a description of a restartable

operation.

This array is updated using normal transactions semantics.

Operation descriptions may be updates and the last step of a complex operation can release the slot.

Recovery of the file system restarts all pending complex operations after the file systems is consistent.

Deleting unlinked files on startup

Unlinked but still opened files need to be deleted after a system failure.

A limited number of slots are used for this purpose. Once more unlinked but open files exists a special restartable operation is used for file deletion on startup. This operations uses an on disk double linked list of the to be freed inodes. The head of the list is stored in the

complex operation slot and the per element pointers re-use fields in the inode not used in unlinked

inodes. This inode list is modified by transactions so list operations are atomic.

Concurrency

The VFS, the file system layer of FreeBSD, currently requires files/directories to be exclusively locked before they are modified. This file system externally enforced locking is sufficient to protect meta data owned on a

per file/directory base.

Per file locks are not sufficient to protect the free inodes and fragments bitmap meta data.

However since allocated blocks/inodes are protected it is sufficient to protect block/inode allocation with a lock. This is done using per cylinder group block locking.

Concurrency (Cont)

When inodes/blocks resources are freed they leave the protection of the per file/directory lock.

To prevent inconsistencies, transactions that free resources must be written to the log before transaction that use the just freed resources.

This is easily accomplished by postponing the freeing of the resources until the intend records of a transaction are written to disk.

Caching

Bluffs bypasses the classic buffer layer and interacts directly with the Virtual Memory (VM) layer .

It participates in the paging decision of the VM layer by implementing a virtual page map.

This virtual page map is used for caching active pages.

As long as pages are `”mapped’” in the virtual page table they can be accessed and their modified and referenced flags can be set without acquiring a page queue or object mutex.

Access to the pages and operations on the virtual page map are synchronized using a hashed mutex.

Pages belonging to the same vnode are divided into fixed size blocks. Each block is then hashed onto a limited number of mutexes.

Caching (Cont)

Special copyin/copyout operations are used to copy data in or out of the kernel.

These release the mutex lock of the virtual page map when a page fault on a user page occurs and restart the operation after the page fault is handled.

Caching (Cont)

Special copyin/copyout operations are used to copy data in or out of the kernel.

These release the mutex lock of the virtual page map when a page fault on a user page occurs and restart the operation after the page fault is handled.

This avoids having to wire or hold pages.

The actual page data is read using ephemeral mapping techniques to avoid the need for interprocessor

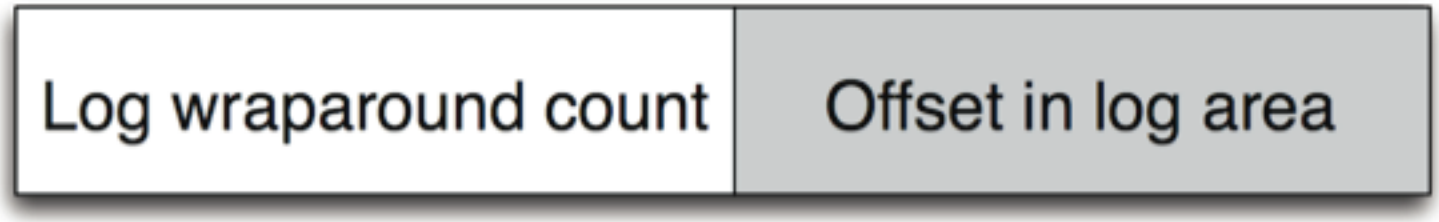
Translation Lookaside Buffer (TLB) shootdown that require expensive interprocessor interrupts (IPIs).

Log Sequence Number (LSN)

A log sequence number is used to uniquely identify log records.

This is used to identify, find and retrieve a log entry on the disk and for reasoning about the order of log records.

Bluffs constructs the LSN, an unsigned 64 bit type, by using lower bits for the offset in the circular log data, and higher bits for a count on how often we wrapped around to the start of the are log area.

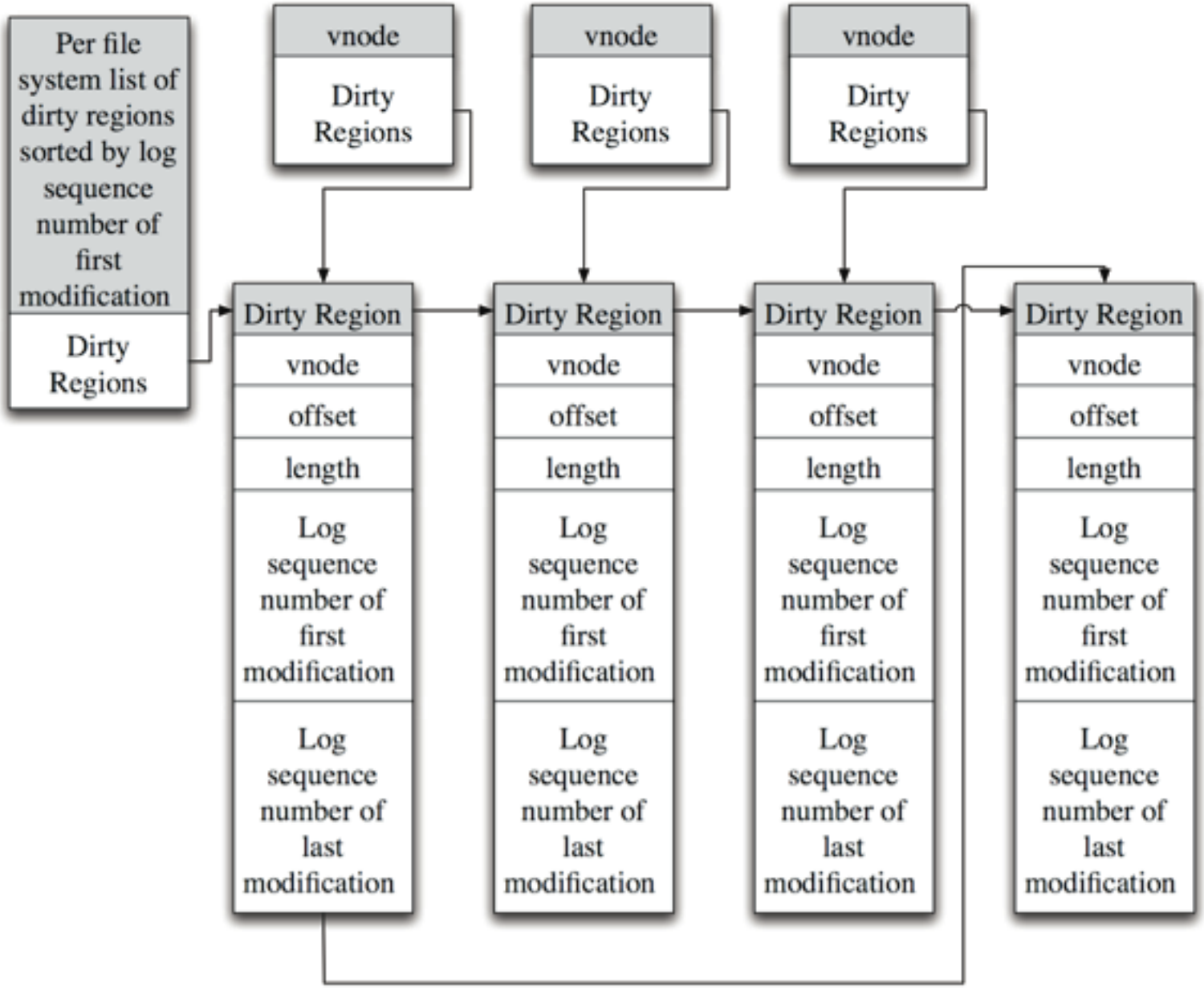


For two LSNs X and Y, X < Y indicates the log entry identified by X was written before the log entry identified by Y

Tracking dirty (modified) meta data disk regions with dirty region descriptors

Bluffs tracks modified data on regions (mostly block sized) while preserving sector update semantics

The dirty region descriptor is one of the central data structures in Bluffs and serves multiple purposes described in detail in the next slides


Per file system list of dirty regions sorted by log sequence number of first modification
Dirty Regions
vnode
Dirty Regions
vnode
Dirty Regions
vnode
Dirty Regions
Dirty Region
vnode
offset
length
Log sequence number of first modification
Log sequence number of last modification
Dirty Region
vnode
offset
length
Log sequence number of first modification
Log sequence number of last modification
Dirty Region
vnode
offset
length
Log sequence number of first modification
Log sequence number of last modification
Dirty Region
vnode
offset
length
Log sequence number of first modification
Log sequence number of last modification

Enforcing WAL protocol with dirty regions

When meta data pages are forced out of memory by the page daemon the put_pages function is called on the owning vnode. (All meta data, including cylinder groups blocks are mapped onto vnodes)

The associated dirty region is looked up.

If the LSN of the last transaction that changed the region indicates that it is stable - the pages can be written to disk directly.

Otherwise a (partial) log flush is forced so that the transaction becomes stable. Once the transaction is stable the pages are written to disk.

Writing checkpoints using dirty regions

The checkpoint record contains dirty regions and the LSN of their first modification.

The checkpoint record is written by iterating through all dirty regions of a file system.

Log truncation using dirty regions

All dirty regions are on a per file system list sorted by oldest modification LSN.

By flushing dirty regions in oldest modification LSN order part of the log becomes obsolete and can be reused once the next checkpoint is written.

Bluffs will start flushing dirty regions using this order once the free space between log head and log tail

Is smaller than a certain threshold.

Flushing and writing a checkpoint record does take time so it must be done early enough so that free space is generated before it is needed.

Otherwise file system operations that write log entries would be stopped until sufficient free log space is available.

Limiting the number of dirty regions

All dirty regions are on a per file system list sorted by the LSN of the last modifying transaction.

By flushing dirty regions in reverse order we can limit the number of dirty regions by flushing out regions that were not recently used.

This limits the size of the checkpoint record while preventing recently used regions to be flushed.

A small checkpoint record means faster recovery since less regions need to be recovered.

Recovery (STEP 1)
Finding the most recent Checkpoint

- Recovery starts by reading the log anchors.
- The most recent log anchor is used as a starting point into the circular log and points to a checkpoint record.
- Starting at this checkpoint record the log is sequentially read and scanned for checkpoint records until the end of the log is reached.

(log anchors may become optional to eliminate a write dependency in which case the whole log is scanned to find the most recent checkpoint)

Recovery (STEP 2)
Updating the checkpoint information

Beginning at the last checkpoint record all intend records from transaction that are stable are read and analyzed.

Intend records that describe regions not in the checkpoint will cause the region to be added to the checkpoint.

Intend records that describe fragments being freed will remove remove the freed regions from the checkpoint.

Recovery (STEP 3)
Recover regions described in the checkpoint

Beginning at the oldest intend record needed by any checkpoint record all intend records from transaction that are stable are read and analyzed.

Intend records that describe regions not in the checkpoint will be ignored since they are not needed for recovery.

All other Intend records will be read and the idempotent operation will be applied to the disk region.

Checkpoint records will be ignored.

Recovery (STEP 4)
Repair circular log

Out of order and incomplete writes may leave sectors ahead of the log head with a valid sequence number for the next block write operations.

Before writing any new log blocks the sequence numbers of these sectors need to be invalided.

Recovery (STEP 5)
Flush all data to disk

Dirty regions recovered during Step 3 are flushed to disk.

Recovery (STEP 6)
Write new empty checkpoint

Write new empty checkpoint since all recovery work is done and the log can be truncated.

Update log anchors if enabled.

Recovery (STEP 7)

Read the complex restartable operation array from disk and restart operations in non empty slots.

The file system is now completely recovered.

Building Transactions

So far we have only talked about consistency of completed transaction.

However transaction need to be build by file system operations and transactions may not complete when the file system operation detects an error.

File system operation view of a transaction

A file system operation uses a transaction structure to build a transaction.

It adds change records to a list contained in the transaction structure.

These change record indicate data changes to an in memory dirty region.

Each change record belongs to single transaction and modifies a single dirty region.

Each dirty region has list of change records that plan to modify the region.

Update in memory

A change record describes a modification to an on disk region. But it is also used to modify the in memory copy of the dirty region.

While on disk regions can not be modified before a transaction is stable the in memory copy of the data can be updated.

The change records support updating the in memory copy of the regions at three different times.

Modifying the in memory region immediately

Change records support modifying the in memory disk region immediately.

This requires the change to be permanently undone if the transaction does not complete or temporary undone if the region needs to be flushed to disk.

A change record contains all the information for undoing and redoing changes.

Modifying the in memory region at transaction completion time

Change records support modifying the in memory disk region once the transaction wrote its intend records to be log buffer.

The change record contains all the information for applying the change.

This is used to prevent race conditions on released resources when reuse can cause race conditions with concurrent transactions.

Modifying the in memory region once the transaction is stable.

Change records support modifying the in memory disk region once the transaction is stable.

The change record contains all the information for applying the change.

This is used to prevent race conditions on released fragments when they are reused as a data buffer and as such do not participate in WAL.

Canceling a transaction

A transaction can be canceled at any time before it completes.

Canceling transactions requires rolling back in memory changes.

This is done by iterating through all change records of a the transaction in their reverse creation order.

Any modification described in an immediate change record needs to be undone.

All change records and the transaction structure are released.

Completing a transaction

Completing a transaction is done by iterating through all change records.

Each change record contains enough information to generate an intend log record that is written to the log buffer. The last intend record will contain an end of transaction flag.

The dirty regions will be updated with new LSN information about the written intend log entries.

The in memory dirty regions is updated if the change record requires it.

Most change records and the transaction structure are released.

Availability

Bluffs ***was*** scheduled to be released for testing in Q1.

-But I had some very good excuses for being late

Bluffs is ***now*** being scheduled to be released for testing in Q2.

- Unless I acquire another set of good excuses