DOCUMENT NO.	REVISION NO.	STATUS
725-xxxx-0001	0.1	**DRAFT**
DEPARTMENT	AUTHOR/CONTACT	EMAIL
Engineering	Anurag Agarwal	anurag.agarwal@onstor.com

Design Specification:

Flexible File System Journal

Approvals:

Name/ Title	ECO

ONStor
Always-On Data

**

Contributors

Name	Email
Anurag Agarwal	anurag.agarwal@lsi.com

Disclaimer Notice

This document is provided “as is” with no warranties whatsoever, including any warranty of merchantability, non-infringement, fitness, for any particular purpose, or any warranty otherwise arising out of any proposal, specification or sample.

All liability, including liability for infringement of any proprietary rights, relating to use of information in this document is disclaimed. No license, express or implied, by estoppels or otherwise, to any intellectual property rights is granted herein.

This document is subject to change without notice, and does not represent any commitment by ONStor, Inc. to develop any product or service based upon the information contained herein.

Copyright Notice

Revision History

Rev	ECO	Written By	Page/Sect	Revision Summary	Date
V0.1		Anurag Agarwal	All	Draft	12/08/09

Table of Content

**

1. Related Documents

Flexible File System Journal Functional Specification document.

2. Problem Statement

This is taken from Flexible File system Journal functional specification document.

Our current file system places the journal at the beginning of the very first LUN in the volume. The file system can grow over time and more efficient LUNs can be added to the volume. Currently, there is no way to move the journal to fast LUNs added to the volume. As the journal is heavily written all the time, if faster LUNs are used for journal, overall performance can be improved. Also having the journal at the beginning of the volume does not produce best results under heavy load, since a lot of the initial metadata of the file system is also placed at the beginning of the volume.

Currently, the file system journal uses 64MB log. This may be not sufficient for metadata intensive workloads. It is desired to have a configurable amount of log space based on the workloads.

3. Solution

This feature will allow resizing of the log as well as relocation of the log.

There will be following new arguments to volume create :

- - b LOGSTARTBLOCK
  If specified, the file system attempts to place the journal after LOGSTARTBLOCK blocks (of size 8192 bytes) from the beginning of the file system. If not specified then log will be placed at default log location, which is after initial meta data in the file system (after block 1026). It should be 1M aligned to match the chunk size of SVM.
- -f LOGLENGTH
  If specified, the file system will allocate LOGLENGTH bytes for the file system journal. Valid values are 64M, 128M, 256M, 512M and 1G. If not specified, default LOGLENGTH is 64M.
- -i LOGLUN
  If specified, the file system places the journal on the specified LOGLUN. There are two choices here:
  - First is to consider LOGLUN part of file system space. Log will be placed at the beginning of the lun. But remaining part of lun will be used for other allocation.
  - Second is to use LOGLUN exclusively for log. Lun will be part of volume, but it will not be part of file system space. For this case the lun specified here must be free and complete lun will be used for log only.

  In both the cases size of log is governed by LOGLENGTH parameter. For time being first option will be implemented. Second option can be explored and implemented along with multi devices support work.

  LOGLUN and LOGSTARTBLOCK are exclusive parameters, only one of them can be specified.

There will be following new arguments to volume modify:

- -b LOGSTARTBLOCK
  If specified, the file system modifies the location of journal from existing location to free space available after the LOGSTARTBLOCK blocks (of size 8192 bytes) from the beginning of the file system. LOGSTARTBLOCK needs to be 1M aligned.
- -f LOGLENGTH
  If specified, the file system modifies the size of journal to new LOGLENGTH bytes. Valid values are 64M, 128M, 256M, 512M and 1G. If sufficient free space is not available at the current location of journal then journal can be moved to other location after the current location.
- -i LOGLUN
  If specified, the file system modifies the location of the journal and places it on the specified LOGLUN.

  LOGLUN and LOGSTARTBLOCK are exclusive parameters, only one of them can be specified.

Volume show command will be modified to show logstart block and log size.

Rules regarding file system journal:

- Journal will be contiguous on lun.
- Journal will not span across multiple luns.
- Default size of journal will be 64M.
- Default placement of log will be after file system block 1026, which is after initial metadata blocks.

- Log location will be 1M aligned.
- If LOGSTARTBLOCK is specified, the file system will try to place the log after that many blocks in the file system where contiguous space available on a single LUN.
- The blocks of the journal are not subject to copy-on-write (COW) even if snapshots are present.
- The journal will not be transferred to the target volume as part of mirror transfer even though the target volume will inherit any modifications to the size and location of the journal in the source volume.
- Modifying the journal should change super block to point to new log and old log blocks should be truncated.
- Valid values for log size are 64M, 128M, 256M, 512M and 1G.
- There is no need to be backward compatible, as this feature is planned for next generation file system part of “Orion”.

4. Overall Design

Following issues needs to be addressed for this feature:

- Converting fix location and fixed size log to variable size and flexible location. Layout of log needs to be changed for this. More details in the detailed design section.
- Managing transition of log from old log to new log. New log needs to be allocated and old log needs to be truncated. It can be tricky with system crashing in between these two operations.
- Large log will require large amount of memory to replay the log. Currently log replay brings the complete log in the memory before replaying. This needs to be changed.

5. Detailed Design

5.1 Disk Layout changes

Currently space for log does not come from the file system space. File system allocation space start from the end of log.

Existing file system layout is:

- 64M at the beginning of volume is reserved for log.
- After that 64M is also reserved. Owner block is kept at the end of this 64M reservation.
- File system space starts at 128M boundary. This offset is treated as offset 0 inside the file system. Super blocks starts at this logical offset 0.

This layout will be changed to following:

- 1M reserved area at the beginning of volume. Size of this reserved area will not be hard coded in the file system. This size will be specified in the first lun label. Changing this reserved area will require changing this value in lun label. Volume manager will pass this value to file system in make file system and mount requests. There is no need for file system to read this value from disk label. This value can not be changed once a volume/file system is created.
- Owner block will be at the end of this 1M reserved space. Size of owner block does not change. Location of owner block will also be coming from lun label. It will not be hardcoded in the file system.
- File system space will start at 1M boundary. Super block will remain at the offset 0 in the file system.
- If user specifies LOGSTARTBLOCK with volume create then LOG will be placed at that location. This location has to be 1M aligned. Otherwise Log will be placed at first 1M boundary after the end of initial meta data. 1M alignment is chosen to match SVM chunk size.
- User can use LOGLENGTH to specify size of the log; otherwise default log size will be 64M.
- If user specifies LOGLUN then first we will figure out the placement of that lun in the file system space, and then file system space will be allocated from the beginning of that lun. LOGLUN is essentially an alternate interface to specify LOGSTARTBLOCK. There will no separate LOGLUN information stored in the super block, only LOGSTARTBLOCK will be stored in super block.

5.2 Create log at volume create

At the volume create time, based on LOGSTARTBLOCK, LOGLENGTH and LOGLUN parameters, space will be allocated from the space map. We will mark corresponding blocks not to be copied on write in the reference count file.

LOGSTARTBLOCK and LOGLENGTH fields will be stored in the super block.

5.3 Modify log

Size and location of the log can be modified using the volume modify command. There are two cases to be handled here:

- Location of log is not changing, only size of log is changing and there is sufficient contiguous space from existing log to grow the log. In that case, extra blocks will be allocated from file system. They will be marked appropriately in the reference count file. And log size will be updated in super block.

- Location of log is changing, either because user has specified a different location, or because contiguous space is not available to grow the log. In this case of location of log changing, new space needs to be allocated, super block needs to be updated and old log needs to be truncated. One needs to be very careful with respect to system crash in the middle of such operation.

Space allocation, free up and super block updates must be transactional. It is difficult to implement all these three operation in a single transaction, because log itself is changing and there are two different logs involved here. It is also difficult to do large allocation/de-allocation operations in a single transaction.

These operations will be divided in two phases, one on the old log and other on the newer log. A new meta inode will also be introduced.

Here are the steps to implement log allocation/log resize operation:

1. Use a meta inode in the file system as a temporary place holder for log space. There is a FS_TEMP_META_INODE, added recently for trashcan feature. This inode will be used with appropriate serialization with trashcan feature.
2. Allocate new space that is to be allocated to log to this inode. This operation can be done without freezing the file system.
3. Mark these blocks not to be copy on write in reference block file.
4. Zero the new log blocks.
5. All these transaction will go to the existing log.
6. Freeze the file system.
7. Reset the existing log.
8. Change the incore data structures to point to new log.
9. Start a transaction in the new log to swap the blocks between old log pointed by super block and new log pointed by FS_TEMP_META_INODE. At the end of this operation, super block will point to new log and FS_TEMP_META_INODE will point to old log.
10. Flush the super block to disk. It is quite important to make sure that on disk super block points to new log, before FS_TEMP_META_INODE starts pointing to old log blocks.
11. Unfreeze the file system.
12. In the new log, free the blocks in FS_TEMP_META_INODE.

Failure handling will be quite simple. Replay the log to get file system to consistent point. FS_TEMP_META_INODE will never share the blocks with log pointed by super block. It will either point to blocks that would have become new log, or it will point to block that were part of old log. It will be sufficient to just free the blocks in FS_TEMP_META_INODE after the mount. Super block will always point to either old log or new log depending on point of crash.

Let's consider various failure scenarios here:

1. If system crashes any point before 8, then super block will point to old log. After log replay, mount will free the blocks allocated to FS_TEMP_META_INODE. It will be effectively aborting the log change operation.
2. If system crashes after committing the transaction, but before updating the super block, step 10, then after the crash super block will still point to old log. And log change operation will be effectively aborted.
3. After step 10. Super block now points to the new log. Log replay will bring FS_TEMP_META_INODE to consistent state. After mount blocks of FS_TEMP_META_INODE will be freed, this will effectively free up the old log blocks.

5.4 Super block changes

Two new fields will be added in the super block. These fields are:

- sbLogLength: Length of the log in terms of file system blocks.
- sbStartLogBlock: Start block of the log, in terms of file system blocks.

5.5 Space allocation and free

There will be explicit checks and asserts added to space allocation and free code to make sure that space allocated to log is never shared with any other files. It will be done to catch software bugs. Similar kind of validation will be added in the eek.

5.6 Log Segments

Currently log is divided in the 64 segments, each of size 1M. There is one bit for each segment in inode to keep track of log dependency. This logic will remain same. Number of log segment will remain 64, but size of log segment will change depending on the size of log.

5.7 Managing replay of large log

Currently, memory requirement for log replay is size of active log plus memory required to keep all the meta data modified during log replay. This works for the current log size of 64M, but it would not scale for the larger log.

5.7.1 Existing Log Replay

Log replay has following phases:

- Log replay finds the head and tail of the log from log header record. Then it checks if the volume log cache has enough space to read the complete log. If there is not enough space in the cache then replay request is queued.
- Reading the complete log in the vol log cache.
- Scan the log to find all the committed log record. This phase is called FS_LOG_ACTION_DONE. This code reads all the log records and marks the committed transaction id in replay bitmap.
- Scan the log again to replay the log record. This phase is called FS_LOG_ACTION_REPLAY. In this phase all the committed transaction records are replayed. Replay brings the meta data in core and applies the modification as specified in the log records to those meta data objects. There are three special log records to keep track of deleted objects. These log records are:
  - FS_LOG_RECORD_INODE_FLUSH
  - FS_LOG_RECORD_BUF_FLUSH
  - FS_LOG_RECORD_QUOTA_FLUSH

  These log records identify corresponding inode/buffer/quota records that have been deleted. If there are any log records for these objects in the intent log prior to removing of these objects, then those records should not replayed. This is achieved in the replay phase by deleting the removed objects from the volume log cache when corresponding flush record is encountered. There is no look ahead in the code to avoid log replay for such objects.

- At the end of replay phase all the modified data in volume log cache are flushed to disk asynchronously. This flushing is done by file system cleaner threads at the end of log replay. Inode and quota cleaner threads wait for access lock. At the end of log replay once file system is thawed, then these threads starts flushing the dirty meta data. Currently cleaner threads are effectively blocked during log replay.

Existing design makes explicit assumption that complete log replay can be first done in core before flushing the changed meta data to disk and hence it avoids a need for multiple passes on the log.

5.7.2 Modifications in the log replay

For large log, it is not possible to do the complete log replay in memory.

First phase of log replay, which identifies the committed transactions in the log, will be extended to identify log flush records. This operation can be called log flush cache operation. It will identify all the objects which have been deleted. There are only three objects requiring this handling, these are inode, buffer and quota records. They have corresponding log flush records to identify the delete event. Log scan phase will keep flush entries with extra state in core. This state will be object id (inode/buffer/quota id) and offset in the log for flush log entry.

Let's call this data structure “log flush cache". This information will be kept in a hash table with hash based on object id and object type.

Log flush cache should not keep entries for transactions that have not been committed. This can be done by adding log flush cache operation as a separate log scan phase after finding all the committed transactions. But it is desirable to avoid adding one extra log scan phase. It can be achieved by following algo:

- Beside log offset, transaction id will be added in the log flush cache.
- For each flush log record a new entry will be added in the log flush cache. That would mean that for same object id, there will be multiple entries in the log flush cache. All the entries for same object will be linked together, sorted on the log offset.
- At the end of first log scan phase, we have information about all the committed transactions in replay bitmap.
- Scan all the entries in the log flush cache, remove all the entries for which transaction has not been committed.
- For each object id, just keep the entry with largest log offset. All other entries can be removed.

Now replay action phase will be as follows:

- While replaying a log record, first check if corresponding transaction is committed, otherwise skip this log record.
- Next check if there is an entry for same object id in the “log flush cache” with later log offset.
- If yes, then don't bring the object in the cache and don't replay the log record for that object.
- If no, then modify the object as per replay record.
- Flushing of dirty meta data will be done by existing cleaner threads depending on the memory pressure. Existing cleaner threads normally wait for access lock, which will be not be available while log replay is in progress. But they have an exception to handle flushing of dirty meta data created by eek. Same logic will be extended to flush meta data created during log replay.
- It will not be possible to flush inode during log replay in the current scheme of things. In order to flush inode, we need to do bmap on the inode. Given that there might be subsequent pending bmap modification records in the log, bmap can’t be done until log replay is complete. This problem needs to be addressed by recording block information for inode in the log. This would eliminate the need for bmap during log replay. This solution would require maintaining block information in the in-core inode, which can potentially change with CoW.

- Quota records do not have above problem, log record for quota already has block number for quota record, and hence it does not require a bmap for flushing.
- Zero fill records needs to be processed after completion of log replay as also requires bmap on the inode.

The extra memory consumption for log flush cache will not be very high. With the above logic, the memory consumption for replaying the large log is not very high.

5.8 Limits of Flexible log

Log size can be 64M, 128M, 256M, 512M and 1G.

Log start needs to be 1M aligned.

5.9 API's for Flexible log

This feature can be used by using new options to volume create and volume modify command.

5.10 Command Interface for This Module

There are three new options added to volume create and volume modify command. These options are described in detail in section 3 of this document.

5.11 Performance Expectations

This feature should improve the file system performance, as more log space would mean more operation can be logged before throttling the file system.

5.12 Testability

The module can be verified with the following test methods and test cases.

5.12.1 Test Methods

Following areas will be focused during testing

- Make sure there is no regression with respect to existing file system operations.
- Create large number of flush records by deleting large number of files and by truncating large number of files and directories. Test eek's ability of handle this.
- Trigger log replay in middle of large operations by forcing volume exception or crash.
- Trigger volume modify in the middle of heavy file system operations.
- Introduce various crash points in the log relocation code and test that replay is able to handle all the scenarios.
- Verify that eek duplicate block also takes log blocks into the account.
- Test log roll over with large log size.
- Create multiple file system with large logs and large number of pending operations in each log. Eek should be able to handle it

5.12.2 Unit Test Cases

All the current file system test cases should be run to verify the file system consistency. The test cases can be found from the following path.

test/t/all/features/filesystem/conformance/functional

Run the stress test with larger log size. In the stress test add the options of changing the log parameters while stress is going on.

5.13 Cluster Interactions

None

5.14 File System Interactions

Volume create and volume modify command can be used to trigger this feature.

5.15 Planned Enhancements

1. Add support for dedicated LUN for the file system log.
2. Having the flexibility to check the owner block without mounting the file systems.