Atomic Recovery Units: Failure Atomicity for Logical Disks

Robert Grimm Wilson C. Hsieh M. Frans Kaashoek
Laboratory for Computer Science, M.I.T.
Cambridge, U.S.A.
{rgrimm, wchsieh, kaashoek}@lcs.mit.edu

Wiebren de Jonge
Dept. of Mathematics and Computer Science, Vrije Universiteit
Amsterdam, The Netherlands
wiebren@cs.vu.nl

Abstract

Atomic recovery units (ARUs) are a mechanism that allows several logical disk operations to be executed as a single atomic unit with respect to failures. For example, ARUs can be used during file creation to update several pieces of file meta-data atomically. ARUs simplify file systems, as they isolate issues of atomicity within the logical disk system. ARUs are designed as part of the Logical Disk (LD), which provides an interface to disk storage that separates file and disk management by using logical block numbers and block lists. This paper discusses the semantics of concurrent ARUs, as well as the concurrency control they require. A prototype implementation in a log-structured logical disk system is presented and evaluated. The performance evaluation shows that the run-time overhead to support concurrent ARUs is negligible for Read *and* Write *operations, and small but pronounced for file creation (4.0%–7.2%) and deletion (17.9%–20.5%), which mainly manipulate meta-data. The low overhead (when averaged over file creation, writing, reading, and deletion) for concurrent ARUs shows that issues of atomicity can be successfully isolated within the disk system.*

1. Introduction

Atomic recovery units (ARUs) are a new mechanism to be used by logical disk systems to implement failure atomicity. They combine several disk calls into a single logical call, and thus provide a larger grain of atomicity to disk clients than a single disk operation. ARUs guarantee that, after a failure, either all or none of the operations within an ARU become persistent. Disk clients such as file systems can use ARUs to atomically update multiple on-disk file system data structures. If these meta-data updates are done within one ARU, either all or none of the updates are guaranteed to become persistent, which greatly improves file system consistency. The support for ARUs on the disk system level also removes complexity from the file system, and ensures that multi-operation atomicity can be implemented efficiently. This paper describes the design, implementation, and performance of ARUs.

ARUs protect clients (e.g., file systems) of the logical disk system against software failures within the client, as well as against hardware failures that affect the operability of the storage system, such as power outages or partial media failures. ARUs guarantee that all disk operations in an ARU are treated as a single operation during recovery. All or none of the disk operations of an ARU are persistent after recovery. An ARU can thus be seen as a light-weight form of transaction. It provides atomicity with respect to failures, but not with respect to concurrency. In particular, clients need to define and implement their own locking mechanisms. ARUs also do not guarantee durability. However, if desired, file systems and databases can easily implement these functions on top of ARUs.

ARUs are designed as part of the Logical Disk [4]. The Logical Disk (LD) is an interface to disk storage that provides its clients with a virtual representation of disk storage by using a logical name-space for disk blocks. File management and disk management can thus be separated, which removes complex disk optimizations from the file system. ARUs further simplify the implementation of file systems. For example, a file system can bundle the allocation of a file, entering the file in its directory, and updating the meta-data for the file and the directory in a single ARU. The implementation of ARUs will guarantee that if a failure happens all or none of the operations happened. Many of today's file systems require specialized recovery procedures and provide ill-defined behavior with respect to failures. With ARUs file systems do not need specialized recovery procedures (such as `fsck`), can precisely define the semantics of file operations with respect to failure, and their implementation can be less complex.

To demonstrate the fact that concurrent atomic recovery units at the disk level perform well, we present a prototype implementation. The prototype is based on the log-structured prototype of LD (log-structured logical disk or LLD, see [4]), which originally did not support concurrent ARUs. We use the same combination of LLD and the Minix file system [17] as in [4], which shows excellent performance on *Write* operations (utilizing 85% of the available bandwidth as compared to 13% for the Minix file system by itself) and good performance on *Read* operations (depending on the workload). We also use the same experiments as in [4] for our performance comparison, which allows us a direct evaluation of the effects of concurrent ARUs on run-time behavior. We demonstrate that the performance difference between a regular disk system (without support for concurrent ARUs) and our prototype (with support for concurrent ARUs) is negligible for *Read* and *Write* operations, while file system operations which require more modifications to disk system meta-data, such as file creation and deletion, incur a small, but visible overhead.

The contributions of this paper are twofold. First, we introduce concurrent atomic recovery units for disk systems, and show that they provide a convenient means to provide a larger grain of failure atomicity than a single disk call. We define the exact semantics of concurrent ARUs and detail the policies regarding concurrency. Second, we present a prototype implementation in the log-structured logical disk system and show that concurrent ARUs can be implemented without incurring a large performance penalty for the added functionality.

This paper is organized as follows. The logical disk system and its log-structured prototype are shortly reviewed in Section 2. The motivation for introducing more advanced semantic capabilities on the disk system level and the semantics of concurrent atomic recovery units are presented in Section 3. The prototype implementation of concurrent atomic recovery units in the logical disk system is discussed in Section 4. The prototype implementation is then evaluated in Section 5. Related work is discussed in Section 6. Finally, Section 7 summarizes the experience of designing and implementing concurrent ARUs within LLD.

2. Background: The Logical Disk System

ARUs are designed as part of the logical disk system (LD). LD separates disk and file management by providing an abstract interface to disk access [4]. It provides its clients with a virtual representation of disk storage through the use of a *logical* name-space for disk blocks. Blocks are the smallest unit of disk storage in LD, and can be further arranged in larger logical units called lists. These ordered lists represent the logical relationship between blocks, and provide a guide to LD for the physical allocation of blocks on disk. The LD interfaces support block read and write operations (*Read* and *Write*), block allocation and de-allocation (*NewBlock* and *DeleteBlock*), and list allocation and de-allocation (*NewList* and *DeleteList*). Blocks are always allocated within a list, and are either placed at the beginning or after a specified predecessor. The *Flush* operation ensures that all data and meta-data have been written to disk.

The logical disk system separates disk and file management, which has three distinct advantages over systems with closely integrated file and disk management. First, it removes complex disk optimizations from the file system, which allows file system designers to focus on the organization of files, not on their layout on disk. Second, a particular implementation of LD can be optimized for the most prevalent disk access patterns, and LD implementations can be exchanged transparently, without changing applications or other parts of the operating system. Similarly, several different file systems can share a particular LD implementation. Third, it allows for efficient solutions to the I/O bottleneck. Since LD presents a virtual representation of disk storage, it can transparently re-organize the layout of data on disk to reduce access times.

The log-structured LD prototype (the log-structured logical disk system or LLD) is modeled after Sprite LFS [11, 12]. It divides the disk into large, fixed-size segments that are filled in main memory and written to disk in single disk operations. Segments are divided into two parts. The first and larger one contains the actual disk block data; the second, called the *segment summary*, serves as an operation log for LLD's own meta-data. The mapping between logical and physical block identifiers, as well as all list information, is contained in the segment summaries, and can be reconstructed during crash recovery by scanning them. To improve performance the same information is maintained in two tables, called the *block-number-map* for the logical-to-physical block mapping and the *list-table* for the lists of blocks. If LLD runs out of disk space it uses a *segment cleaner* to reclaim unused disk space.

3. Atomic Recovery Units

An important criterion for the design of a disk system interface is the support for disk operations which have larger granularity than a single disk call. This criterion is based on two realizations. First, file systems have a legitimate need for a larger grain of disk system granularity when making updates to files, particularly when updating meta-data structures (e.g., when creating files in directories). Providing failure atomicity for these operations on the disk system level reduces file system complexity in general, improves file system consistency and therefore also reduces the complexity and length of the crash recovery process (e.g., as compared to the UNIX `fsck` utility). Second, failure atomicity over several disk operations is necessary to efficiently support transaction-based systems as direct disk system clients. Current transaction systems are often implemented on top of Unix-like file systems, or they bypass the file system altogether and utilize the raw disk interface. The mapping of stricter transaction semantics in these systems onto more lenient file or disk system semantics implies an undue performance penalty (mainly due to synchronous writes and excessive buffer flushing). A disk system design should thus simplify system complexity and improve efficiency by providing support for atomicity.

To address the above criterion we provide the *atomic recovery unit*:

> **All disk operations in an atomic recovery unit are treated as an indivisible operation during recovery: the disk system guarantees if a failure happens that all or none of these operations remain persistent after recovery.**

The LD interface specifications support ARUs through the use of the *BeginARU* and *EndARU* operations. Operations that are not part of an explicit ARU are atomic and are called *simple* operations. When creating a file, a UNIX-like file system would create the file's i-node and change its directory's data within the same ARU, which would guarantee that after a failure either the file has been properly created and is accessible through its directory (with all meta-data structures being persistent), or that none of the changes will be persistent. As already pointed out, file system consistency is greatly improved by this atomic file creation, which simplifies consistency checking. In contrast, conventional file systems usually use some meta-data update sequencing strategy—by using synchronous writes and by ordering *Write* operations according to semantic dependencies—which implies an undue performance penalty.

ARUs protect disk system clients against the effects of two classes of failures. First, they protect disk system clients against the effects of software failures within the client itself. Second, they protect disk system clients against the effects of failures that affect the operability of the overall storage system, such as power outages or (partial) media failures.

3.1. Block Versions

We view *Write* operations as a stream of blocks written to disk. By viewing disk access as a stream of blocks, we emphasize that all operations within a particular stream are serialized and executed *in that order*. This order should always be preserved by the disk system and is determined by the time of an operation. Atomic recovery units increase the logical grain from a single disk operation to several disk operations. They are marked in the stream by explicit *BeginARU* and *EndARU* operations. ARUs are serialized by the time of the *EndARU* operation. Ending an ARU will also be referred to as "committing" that ARU.

Since the stream of blocks spans both system memory and disk storage, and since operations can either be part of an atomic recovery unit or be atomic operations by themselves, a data block can simultaneously exist in several versions in the overall system. We distinguish the following three classes:

- The *persistent* versions. The ARU to which a data block in this class belongs has been committed, and the record of this *Commit* operation is in persistent storage; it has been flushed to disk. In an order-preserving stream this statement implies that the data block itself is in persistent storage, as are all other data blocks belonging to the same ARU. Simple operations are persistent once they are flushed to disk.

- The *committed* versions. The ARU to which a data block in this class belongs has been committed (or, if it was a simple operation, the operation itself has finished) but has not been flushed to disk yet.

- The *shadow* versions. The ARU to which a data block in this class belongs has not been committed yet. Simple operations do not generate shadow versions, since they commit as soon as they end.

It is possible to envision a system that provides *Read* access to several different versions of the same block within the same class, especially to several shadow versions within the same ARU. We find these most general semantics excessive and favor overall system simplicity. Thus, we are only interested in the *most recent* version of a block in each class, as determined by the time associated with each operation. For brevity we call this most recent version of a block in each class the persistent, committed and shadow version, respectively. We also refer to the totality of all blocks in any of the three versions as the persistent, committed and shadow states, respectively.

The shadow version of a disk block is local to an ARU and makes the transition to committed version on an *EndARU* operation. During this transition the shadow version either replaces the current committed version (if the shadow version is more recent) or it is discarded. The committed version of a disk block is logically complete but not yet in disk storage. It makes the transition to persistent version on a write to disk (e.g., on a *Flush* operation). During this transition the committed version either replaces the current persistent version (if the committed version is more recent) or it is discarded. The shadow and the committed versions of a disk bock will thus progress within the stream of blocks under normal disk system operation and eventually make the transition to the persistent version. Persistent, committed, or shadow versions can only be replaced or discarded during such a transition. The three different versions in the single stream case and their relationship are illustrated in Figure 1.

Recovery after a failure is always to the most recent persistent version (and not to the most recent committed version nor the most recent shadow version). The shadow state represents ARUs that have not committed yet; these ARUs have outstanding operations at the time of the failure and thus must not be recovered. The committed state represents ARUs that have committed but have not been flushed to disk yet; not all operations of such ARUs are guaranteed to be in persistent disk storage and thus the committed state must not be recovered. The persistent state represents ARUs which have committed and have been flushed to disk; all operations of such ARUs are in persistent disk storage and are thus recovered.

While the above discussion is in terms of disk blocks, it applies to block lists as well.

3.2. Concurrent Atomic Recovery Units

In order to support multi-threaded file systems or several independent clients on top of the disk system concurrent streams are necessary. Each stream supports the same data structures and operations as in the single-stream case; i.e., data blocks and lists of blocks that need to be allocated and de-allocated. Each stream also supports a larger logical grain through atomic recovery units, which results in concurrent atomic recovery units. The block versions, as defined above in Section 3.1 for the single stream case, also generalize for the case of multiple, concurrent streams.

Since disk storage is a shared resource, it is necessary to merge the concurrent streams into a single, merged stream before writing them to disk storage. Simple operations are ARUs by themselves and directly operate on the single, merged stream. The operations of a concurrent atomic recovery unit can be merged into the single stream once the ARU commits. Figure 2 illustrates the three different versions for concurrent streams and shows how the concurrent streams are merged. Each ARU has its own shadow versions. They make the transition to the committed state on an *EndARU* operation and are also merged into a single stream on this transition. Simple operations directly affect the merged stream, and the disk blocks they affect become committed upon completion. All committed versions make the transition to persistent state on a flush to disk. The individual operations, including the *NewBlock* and *NewList* operations, which show slightly different semantics, are discussed in detail in Section 3.3.

3.3. Version Semantics

Given the fact that a block can be in any of the three states discussed above and that shadow versions can exist for each currently active ARU, there can be

$$max(\text{versions}) = n + 2$$

where n is the number of currently active ARUs

different versions of the same logical block which need to be maintained by the disk system. It is thus necessary to define the exact semantics of versions and of the three states (shadow, committed and persistent).

Within the following discussion the version to be accessed may not always exist. For example, if the semantics demand an access to the shadow version of a block, but this specific block has not been written within the current ARU, the shadow version of the block does not exist. The disk system must therefore check for the committed version and apply modifications to a copy of the committed version (if it exists). If the committed version does not exist, the disk system applies modifications to a copy of the persistent version, which is guaranteed to exist. The modified copy of the committed or the persistent version then becomes the new shadow version. The version semantics thus define the starting point of a standardized search, which always works from shadow to committed to persistent version.

Generally, all disk operations executed within an ARU affect the state of that ARU. Thus, data blocks (and lists) modified by these operations result in shadow versions. At the end of an ARU (i.e., after the *EndARU* operation) all operations of the ARU make the transition from shadow to committed version. The committed version is changed as the result of such a transition and as the result of a simple operation (which affects the committed state directly). The persistent version is only changed if all operations belonging to an ARU have been successfully written to persistent storage. In other words, the persistent version is only changed as the result of an explicit (call to *Flush* operation) or implicit (block cache is full) flush to disk, after which the transition from committed to persistent state is made.

Allocating a new block or a new list are the two exceptions to the above semantics. The allocation of the block or the list is always done in the merged stream and is immediately committed (even when executed as part of an ARU) to avoid conflicts when several concurrent ARUs allocate the same block or list identifier. However, the insertion of the newly allocated block into a list is done as part of the concurrent stream and thus within the shadow state (for ARU operations). As a result, simple operations and other ARUs do not see the allocation of the block (it is not part of any list) but at the same time cannot allocate the same block (because it has already been marked in the committed state, even though its allocating ARU has not committed yet).

Figure 1. The three versions in the single stream case.

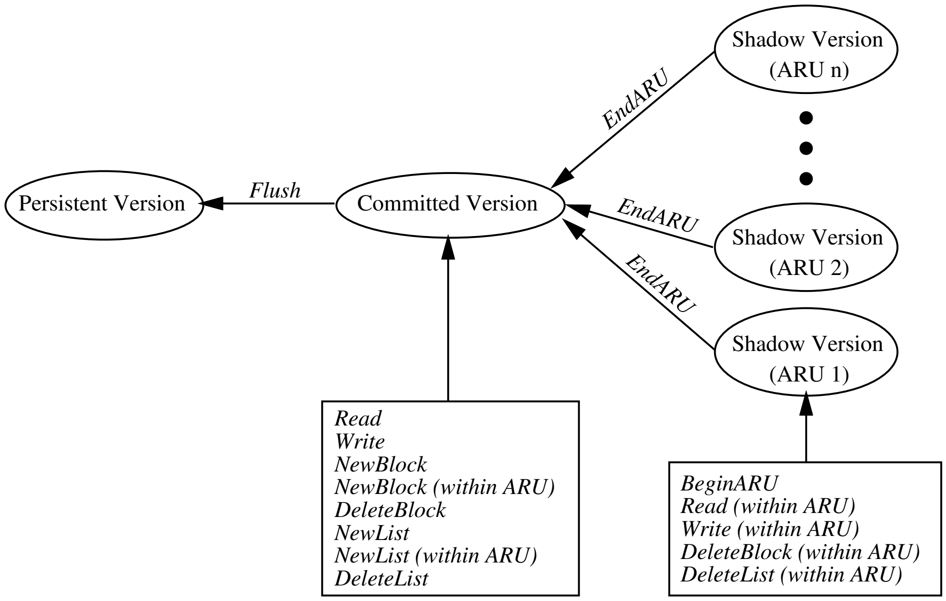

Figure 2. The $n + 2$ different versions in the case of concurrent streams and how they are affected by disk system operations.

The semantics of *Read* operations play a crucial role for concurrent atomic recovery units, because they specify the visibility of operations and thus the degree of isolation between concurrent ARUs. We identify three different possibilities, listed with increasing semantic strength (and thus providing an increasing degree of isolation between the state of concurrent ARUs). The first option always returns the most recent shadow version of all concurrent ARUs; in other words, any update is visible to all disk system clients right away. The second option always returns the committed version; updates are only visible once the corresponding ARU commits (or, in the case of simple operations, that operation finishes). Finally, in the third option, *Read* operations follow the general semantics of *Writes* in concurrent ARUs, in that they return the corresponding shadow version within an ARU and the committed version for simple operations. In other words, the shadow state of an ARU is strictly local to this ARU, visible within the ARU and isolated from the state of all other concurrent ARUs. All updates within this shadow state become visible in one atomic operation after that ARU commits.

The first and second options have the advantage that all disk system clients see the same state. However, this may also be a disadvantage if this most recent state is in fact part of a not yet committed ARU (and thus not persistent). The third option does not differentiate *Read* operations from other operations, reduces the number of special cases and thus improves the consistency of our semantics. Since the shadow states of each ARU have to be completely isolated from each other, it is also the most complex to implement, thus providing a good test case to evaluate the overhead of concurrent ARUs. It has been chosen for our prototype implementation. It is important to emphasize that while the three options specify the visibility of operations, they do *not* imply any concurrency control (such as locks) for *Write* operations. Concurrency control has to be defined and implemented by disk system clients.

If there exist any ARUs that have either not yet committed or whose commit operation has not been written to disk upon recovery, the disk system undoes their operations by clearing all internal state from the effect of these operations. Any data blocks that have been written to disk as part of such an undone ARU are effectively treated as shadow versions and need to be cleaned (that is, to be removed from the set of accessible blocks). Thus, recovery is always to the most recent persistent version. However, if blocks were allocated within such a non-committed ARU they will remain allocated. This is a consequence of the above rule that blocks are always allocated in the committed state. A disk consistency check during recovery should free such blocks (which adds very little overhead to a log-based recovery procedure).

Figure 2 summarizes which operation affects which state. Concurrent streams or shadow states are created by the *BeginARU* operation and all consequent *Read*, *Write*, *DeleteBlock* and *DeleteList* operations within the corresponding ARU affect this shadow state. As already discussed, the *NewBlock* and *NewList* operations within an ARU affect the committed state instead of the shadow state. The concurrent streams are merged into one stream (all shadow state makes the transition to committed state), if the ARU commits through the *EndARU* operation. Simple operations (*Read*, *Write*, *NewBlock*, *DeleteBlock*, *NewList* and *DeleteList* outside an ARU) affect the merged stream or committed state, since they represent ARUs by themselves. On a *Flush* operation all committed state is written to persistent storage and thus makes the transition to persistent state.

4. Implementation

This section gives a general overview of the implementation of concurrent ARUs within LLD. It highlights changes to the initial LLD prototype (without concurrent ARUs) and illustrates the internal administration of the shadow, committed and persistent states.

The mapping between logical and physical block identifiers and all list information is contained in the on-disk segment summaries. LLD also uses two tables (the block-number-map and the list-table) that store this same information to improve performance. For each logical block the block-number-map contains a record that maps the logical identifier to physical address and segment number; it also records the state (allocated or not allocated), the position within a list (i.e., the successor) and the time-stamp for the time when the block was last written. The list-table records the first (and last) block of each list. These tables describe the persistent state and are the same as in the initial version of LLD. Figure 3 (which is the same as Figure 2 in [4]) illustrates these two tables. It also illustrates the layout of disk storage, which is fragmented into segments. Segments are further divided into two parts, one part that stores the data blocks and one part that stores the segment summary.

The tables for the persistent state (which are kept in memory and on disk) are augmented by in-memory singly-linked lists of alternative records describing blocks and lists in the committed and shadow states (one list of blocks in the committed state and one list of blocks for each currently active ARU, and analogously for lists). A block or list record is a member of such a list only if it differs from the record with the same logical identifier in the persistent state. This ordering by state facilitates the efficient transition from one state to the other. However, the access to alternative versions of a block or list record is inefficient. It has been improved by providing a second set of lists with all alternative versions of a block or list record on one list (i.e., all block or list records on such a list have the same identifier). Each block or list record is thus a member of two in-memory singly-linked lists, one for all records in the same state, and one for all records with the same logical identifier. The resulting mesh of block and list records makes both *Lookup* operations by state and by logical identifier efficient.

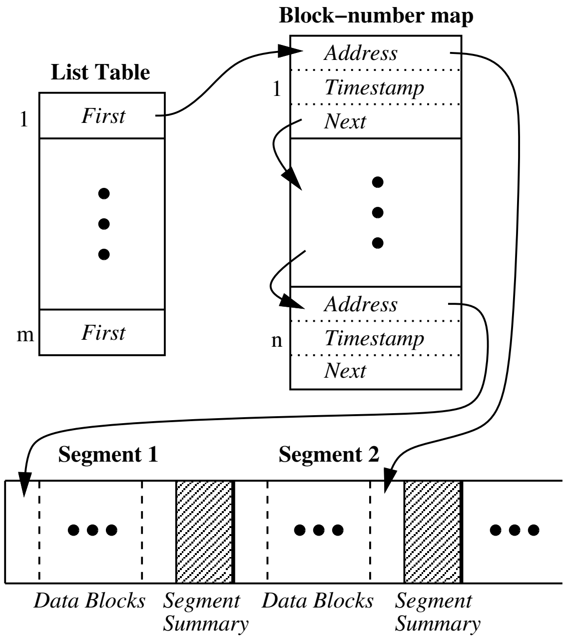

Figure 3. The data structures of the persistent state and their relation to disk layout. This illustration shows the persistent state for logical block 1 which is the first block on list 1.

When inserting a block into a list, LLD updates the corresponding list and block records within the right state (shadow state for operations within an ARU and committed state for simple operations). LLD also needs to record this insertion in the segment summary to facilitate failure recovery. For these purposes LLD generates two link records, one for the link predecessor–block, and one for the link block–successor. These link records are then used during recovery to restore a list to its state before failure. However, in the current semantics concurrent ARUs may cause different versions of the same list to exist, within the shadow state of each ARU. On commit these versions have to be merged to form a single, consistent list. If LLD generated link records in the segment summary for block insertions within an ARU, merging different versions of the same list would be hard, if not impossible, and the segment summary might contain inconsistent list information.

Consequently, to facilitate merging list operations on transition from the shadow to the committed state, an in-memory list operation log has been added to LLD. Every list operation is executed within the shadow state first, but *no* segment summary entries are generated (to avoid inconsistent list information). A log entry (of the form insert-block-after-predecessor) is added to the log of list operations for the specific ARU. On commit the operations in the list operation log are executed again (in the same order as before), but in the committed state, and the correct segment summary entries are generated. After re-executing all list operations and generating the segment summary entries the commit record for the ARU is generated.

The use of the additional in-memory data structures and their relation to disk storage is illustrated in Figure 4. The illustration shows the shadow state of ARU *i*, after the logical block *k* has been allocated as the first block on list *j*. Each ARU has its own data structure (the top left element in Figure 4) with pointers to the beginning of the lists of alternative block and list records in this same shadow state. The lists of alternative block (or list) records in the same state are maintained using the pointers labeled "Next (Same State)." The lists of block (or list) records with the same logical identifier are maintained using the pointers labeled "Next (Same ID)" and originate from the corresponding block and list records in the block-number-map and list-table (see Figure 3). The link log is implemented using a simple singly-linked list, again originating from the ARU's data structure.

The arrangement of data structures is similar for the committed state, except that there is a single merged stream of blocks. As a result, the maintenance of a link log is unnecessary, and the list of block and list records is maintained directly, without an ARU record.

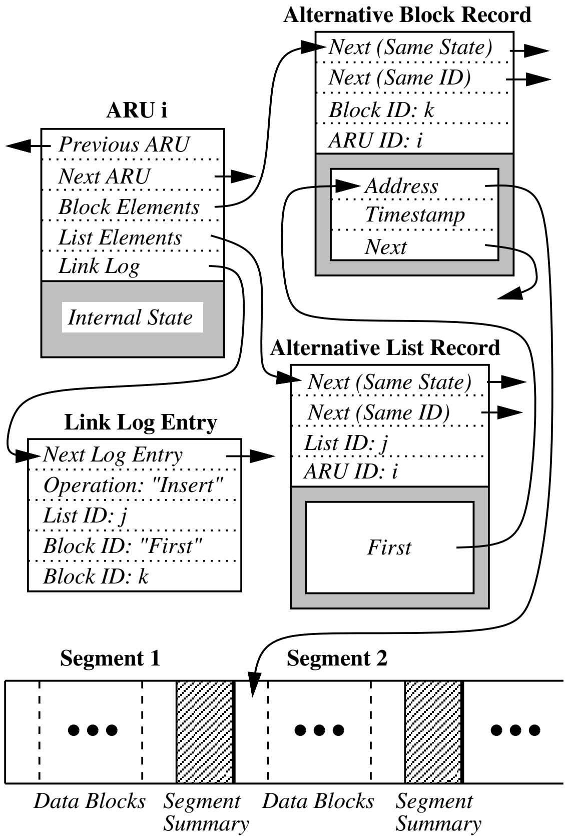

Figure 4. The in-memory data structures of the shadow state and the disk layout. This illustration shows the shadow state for ARU *i* after the logical block *k* has been allocated as the first block on list *j*.

5. Evaluation

This section describes the performance of our prototype implementation. It is structured as follows. Section 5.1 describes how ARUs are used in our prototype. Section 5.2 describes the benchmarks and the system used in our performance evaluation. Section 5.3 discusses the results of the benchmarks and identifies the concurrency overhead in the new version of LLD. Section 5.4 concludes the performance evaluation.

5.1. Using ARUs

The logical disk separates disk management from file management. It supports several independent clients on top of the disk system. Because LD supports concurrent ARUs, each of these file systems may be multi-threaded. Furthermore, the log-structured implementation can be replaced with a different implementation without changing disk system clients. As in [4], we use a single-threaded Minix file system on top of LLD for our experiments (the combined system is called MinixLLD). Since LLD provides the disk management, most of the disk management code (350 lines) has been deleted from Minix.

We modified Minix to perform all directory and file creation as well as all file deletion in ARUs, bracketed by *BeginARU* and *EndARU* operations. Each file is thus created and deleted within its own ARU, which groups both modifications of the file's i-node and its directory's data into one logical unit. After a failure, all or none of the Minix meta-data describing each file will be persistent. It is thus unnecessary to use `fsck` after a failure to restore the file system to a consistent state. Once the disk system recovery is complete, the file system is already guaranteed to be in a consistent state.

5.2. Experimental Environment

The goal of our performance measurements is to identify the run-time overhead of concurrency support. We thus compare the Minix file system running on top of the new version of LLD (with support for concurrent ARUs) and on top of the initial version (without support for concurrent ARUs). Both LLD and Minix are linked together in a single user process that accesses the disk through the raw disk interface provided by SunOS. The performance of the original prototype of MinixLLD (without concurrent ARUs) compares favorably with the Minix file system by itself (for detailed performance results see [4]). MinixLLD shows excellent performance on *Write* operations (utilizing 85% of the available bandwidth as compared to 13% for the Minix file system by itself) and good performance on *Read* operations (depending on the workload).

We repeat the micro-benchmarks from [4]. The first benchmarks measure small file I/O by creating and writing, then reading and finally deleting 10,000 1-KByte files and 1,000 10-KByte files. The second benchmark measures large file I/O. This test uses a 78.125 MByte file, which is first written sequentially (write1), then read sequentially (read1). The file is then written in random order (write2), read in random order (read2) and finally read in sequential order (read3). In addition to these micro-benchmarks, we execute a third benchmark that simply starts and ends an ARU 500,000 times.

We expect the read/write performance of the new version of LLD, especially for large reads and writes, to be comparable to the read/write performance of the old version. Operations that mainly allocate or de-allocate blocks and lists should incur some run-time overhead relative to the old implementation, since the algorithms for manipulating meta-data are more complicated. We expect the overhead to be relatively small, due to the organization of internal data structures in two sets of perpendicular lists.

Our performance measurements are carried out on a 70 MHz SPARC-5/70 workstation with 80 MByte main memory running SunOS 4.1.3. We use a disk partition of 400 MByte on a 2 GByte disk (HP C3010: SCSI-II, 5400 rpm, 11.5 msec average seek time). The SPARC workstation is networked, and no other users are running during the experiments.

5.3. Concurrency Overhead

The three different versions of LLD which are used to determine the concurrency overhead are listed in Table 1. "old" stands for the original prototype of LLD, which supports non-concurrent ARUs and is described in [4]. "new" is our prototype implementation of concurrent ARUs within LLD and "new, delete" is the same prototype implementation of concurrent ARUs within LLD but with improved file deletion within Minix (see below). The partition size for these tests is 100,000 4 KByte blocks (or 400 MByte). Blocks are written to disk in 0.5 MByte segments.

old	The original version of MinixLLD (with sequential ARUs).
new	The new version of MinixLLD (with concurrent ARUs).
new, delete	The new version of MinixLLD with improved file deletion in Minix.

Table 1. MinixLLD versions used in performance evaluations to determine concurrency overhead.

Figure 5 under the "old" and the "new" headings presents the performance results for the small experiments. The new version of MinixLLD differs from the original version in that directory and file creation and deletion are bracketed by *BeginARU* and *EndARU* operations. As a result each file is created within its own ARU and deleted within its own ARU. Experimental results are the average of 10 experiments conducted for each operation. The old version of LLD shows better performance for creating and for deleting the small files, and almost the same performance for reading the files.

The difference between the old and the new version of LLD for creating and writing 10,000 1-KByte and 1,000 10-KByte files amounts to 7.2% and 4.0%. It can be explained by the additional overhead of block allocation in the committed state and transition of block data (for the file system meta-data) from shadow to committed state after the *EndARU* operation. Further overhead is caused by the transition of block allocations and block data (both for the file system meta-data and the actual file data) from committed to persistent state after the *Flush* operation. In this context MinixLLD uses one list per file, which results in the creation of 10,000 lists with one block for the 10,000 1-KByte files and of 1,000 lists with three blocks for the 1,000 10-KByte files. The new version of LLD is slightly more efficient for reading small files. However, the closeness of the results suggests that both versions of LLD will perform equally well for *Read* and *Write* operations, especially for large amounts of data.

The difference between the old and the new version of LLD for deleting 10,000 1-KByte and 1,000 10-KByte files is considerable (24.6% and 25.5%, respectively). Again, as for creating the small files, the block data (for the file system meta-data) is first written within the shadow state and makes the transitions from shadow to committed state after

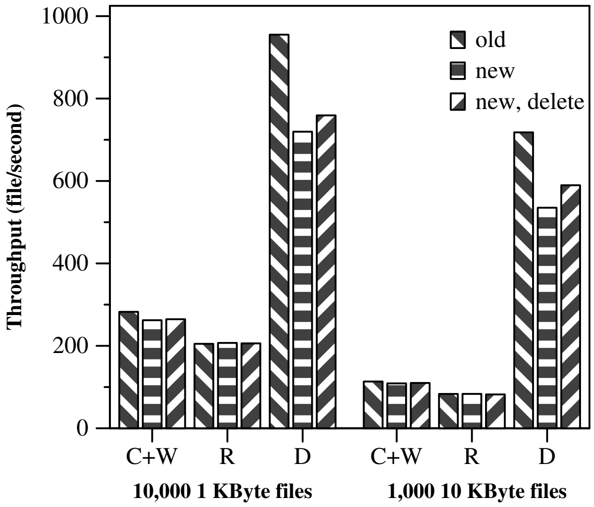

Figure 5. Performance results in file/second for creating and writing (C+W), reading (R), and deleting (D) 10,000 1-KByte and 1,000 10-KByte files. Higher columns are better.

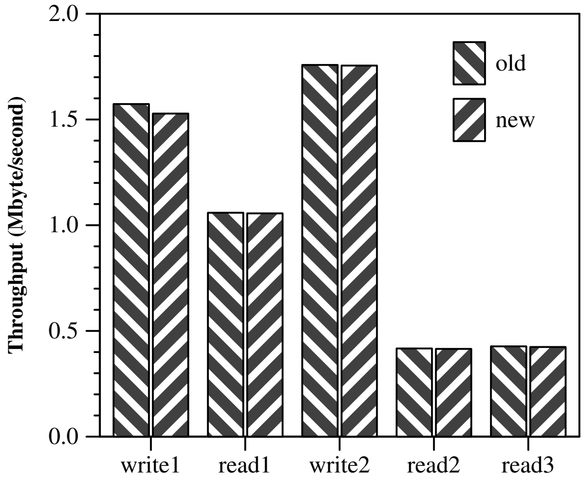

Figure 6. Performance results in MByte per second for throughput of the large experiment. Higher columns are better.

the *EndARU* operation and from committed to persistent state after the *Flush* operation. However, while block and list allocation is always done within the committed state (see the version semantics in Section 3.3), block and list de-allocation is executed within the shadow state. As a result, file deletion creates more LLD meta-data within the shadow state than file creation and thus takes longer for the transition from shadow to committed state after the *EndARU* operation.

Part of the overhead for deleting files in our prototype implementation is caused by the way in which blocks are deallocated in MinixLLD: MinixLLD first deallocates all blocks belonging to a file, which revokes them from the list that represents the file. After deallocating all blocks, it deallocates the already emptied list. Removing blocks from a list requires LLD to search for their predecessors in the list. This predecessor search is one of the causes for the large performance penalty in the new version of LLD; the difference is slightly more pronounced for the 10-KByte experiments, which use longer lists for each file (longer lists cause longer predecessor searches). One possible optimization for MinixLLD is to simply delete the list, without first deallocating all blocks from the list. LLD could then delete all blocks from the beginning of the list, therefore running the expensive predecessor searches less often.

We changed the prototype implementation of MinixLLD to reflect this change in file deletion policy. The results of the experiments using the improved version of MinixLLD can be found in Figure 5 under the heading "new, delete." The deletion of 10,000 1-KByte files shows a clear performance gain, reducing the percent-difference from 24.6% to 20.5%. The performance gain for the deletion of 1,000 10-KByte files is more pronounced (as predicted above) and reduces the percent-difference from 25.5% to 17.9%.

The results in Figure 6 represent the throughput of the large experiment and are in MByte/second. The results represent the average of 10 experiments conducted for each operation. The percent-difference between the old and the new prototype of MinixLLD amounts to 2.9% for first writing the 78.125 MByte file (write1); all other operations show a minimal percent-difference (0.2%–0.7%). The results for the large experiment thus confirm that both the initial version of LLD (without support for concurrent ARUs) and the new version of LLD (with support for concurrent ARUs) indeed show almost the same performance for *Read* and *Write* operations.

When simply starting and ending an atomic recovery unit 500,000 times (by using the *BeginARU* and *EndARU* operations), we achieve a latency of 78.47 μsec per ARU. 24 segments (recording the commit record of each ARU in the segment summary) are written as part of this experiment. The experiment confirms the above interpretation of the results for the small file experiments. ARUs by themselves only introduce a small overhead. The transition of block and list data and of block and list de-allocations from shadow to committed state determines most of the overhead incurred by concurrent atomic recovery units. Furthermore, as our experiment with improved file deletion shows, the order of operations also influences this overhead.

5.4. Summary of Performance Evaluation

Our performance evaluation confirms our hypotheses. Both the old and the new version of LLD show comparable read and write performance, especially for larger reads and writes. We observe a small but noticeable performance penalty in the new version for operations which manipulate large amounts of LLD meta-data. We expect the average overhead of concurrent ARUs over the old LLD prototype to lie approximately half-way between the overhead of file creation (4.0%–7.2%) and file deletion (17.9%–20.5% with improved file deletion). As the new version with improved file deletion shows, LLD clients should pay careful attention to the way they utilize lists and blocks in order to minimize the run-time overhead for meta-data manipulation.

The small overhead of concurrent atomic recovery units in the log-structured logical disk system demonstrates that issues of failure atomicity can be successfully isolated within the disk system. Disk system clients do not need to provide their own mechanism for failure atomicity and their design and implementation can thus be simplified. The good performance of our prototype system is partially a result of the log-structure of disk storage. Other implementations of the Logical Disk will have to utilize at least a meta-data update log to achieve similar performance and to fully support multiple shadow states.

6. Related Work

ARUs are unique in that they allow for clearly separated disk management and file management. Most other approaches to failure atomicity provide this support implicitly (as in file systems) or bundle it with transactions.

Transactions [8] are commonly used in database systems such as System R [1] and Postgres [16]. A successful implementation provides a correct way to group several operations in one larger, atomic unit. It ensures that the effects of concurrent transactions are isolated from each other by some locking mechanism and that the effect on the overall state (e.g., on the database) is persistent. A typical storage manager uses some form of write-ahead log, which records all changes (logging both the state of the database "before" and "after" the change) before they are actually executed on the database. This change log allows both redoing and undoing changes during recovery. Atomic transactions have much stricter semantics than ARUs, are much more heavy-weight than ARUs, and therefore are not appropriate for disk systems.

File system meta-data plays a crucial role in ensuring file system integrity. Conventional file systems, such as the Berkeley FFS [10], use costly synchronous writes to ensure that meta-data updates reach persistent storage. Because of the considerable performance degradation caused by synchronous writes, Ganger and Patt [7, 6] introduce a special form of delayed write, called "soft update." Soft updates serialize meta-data updates to reflect the correct semantic dependencies as blocks are written to disk, which enables the use of delayed writes, but still uses conventional overwrite semantics.

Both the re-implementation of the Cedar File System FSD [9] and the Episode File System [3], use a write-ahead log for all updates to meta-data. By logging meta-data updates before they are executed on the cached disk blocks, the file system avoids unnecessary writes (to force these updates to disk), which results in a more robust file system and a considerable speed-up over comparable file systems without meta-data change logs. FSD further optimizes the write-ahead log by using a group commit protocol, where several meta-data updates are grouped together in one log entry.

FSD, the Episode File System, and the soft update technique illustrate that the overhead of synchronous writes for meta-data updates can be eliminated while at the same time improving overall consistency of the file system. However, these systems provide a specialized and limited solution only for meta-data updates. All three file systems could still benefit from atomic recovery units, which are available for *all* file system operations and not limited to meta-data updates. Furthermore, as our performance evaluation shows, concurrent atomic recovery units are an efficient solution that removes considerable complexity from the file system.

The Mime disk storage architecture [2] is based on the Loge [5] disk controller (which uses internal indirection and shadow-paged block updates) and features atomic writes over multiple blocks through the use of "visibility groups." A visibility group presents a provisional view of disk storage that is isolated from other visibility groups; changes to the disk are only visible within their associated visibility groups. They become visible atomically once a visibility group finishes. All operations of a visibility group can also be aborted. Furthermore, visibility groups support "barriers" which provide a checkpoint for recovery. A visibility group will be restored up to the latest barrier point after a failure. Visibility groups address the same problem as ARUs, but are more complex than ARUs, since they support checkpoints and unrolling, whereas ARUs do not.

The Sprite Log-Structured File System [12, 11] introduced the idea of using a log-like structure to organize disk storage. The original prototype of LLD (without concurrent ARUs) is largely based on the same log-structured design principle; a detailed comparison between Sprite LFS and LLD can be found in [4]. While Sprite LFS shares the general design principle with our prototype implementation of concurrent ARUs within LLD, it does not provide any support for a larger grain of operations or embedded transactions, nor does Sprite LFS feature a clear separation between disk and file system.

Seltzer [15] presents both a user-level transaction system and an embedded transaction system using the Sprite Log-Structured File System. Her embedded implementation features full transaction support at the file system level. Service disruption because of segment cleaning and poor sequential read performance caused her to redesign the log-structured file system, this time as part of the Berkeley BSD 4.4 release. The resulting file system [13, 14] does not support embedded transactions and, by treating the original fast file system and the log-structured file system as alternatives, neglects to specify the semantics of disk access or to introduce more advanced semantics (though it does present alternative optimizations to address different disk workloads).

7. Conclusion

We showed that concurrent atomic recovery units provide LD clients with an elegant and powerful mechanism for failure atomicity over multiple disk operations and for supporting advanced storage semantics (such as transactions). The support for advanced storage semantics is sufficient to efficiently provide atomicity for concurrent disk operations. However, full data isolation and mechanisms for durability must be provided by the disk system clients. The semantics of disk access with concurrent atomic recovery units are made explicit in this paper. They can be taken into account when designing disk system clients, which simplifies the design and implementation of advanced storage semantics on top of the disk system.

The performance evaluation of our prototype implementation within the log-structured logical disk system shows that the concurrency overhead is negligible for *Read* and *Write* operations, and small but noticeable for workloads which require significant modifications to disk system metadata. However, our experiments (especially our changes to file deletion within Minix) also highlight that how data is organized into blocks and lists of blocks influences the performance penalty due to concurrent atomic recovery units.

A source distribution of our prototype implementation is available through the world-wide web at: `http://www.pdos.lcs.mit.edu/ld/`

Acknowledgments

We thank Dawson Engler, Gregory Ganger, Massimiliano Poletto and Deborah Wallach for their technical advice and comments on drafts of this paper.

References

[1] D. D. Chamberlin, M. M. Astrahan, M. W. Blasgen, J. N. Gray, W. F. King, B. G. Lindsay, R. Lorie, J. W. Mehl, T. G. Price, F. Putzolu, P. G. Selinger, M. Schkolnick, D. R. Slutz, I. L. Traiger, B. W. Wade, and R. A. Yost. A history and evaluation of System R. *Readings in Database Systems*, pages 54–68, 1988.

[2] C. Choa, R. English, D. Jacobson, A. Stepanov, and J. Wilkes. Mime: a high performance parallel storage device with strong recovery guarantees. Technical Report HPL-CSP-92-9rev1, Hewlett Packard, 1992.

[3] S. Chutani, O. T. Anderson, M. L. Kazar, B. W. Leverett, W. A. Mason, and R. N. Sidebotham. The Episode file system. In *Proceedings of 1992 Winter USENIX*, pages 43–60, 1992.

[4] W. de Jonge, M. F. Kaashoek, and W. C. Hsieh. The logical disk: A new approach to improving file systems. In *Proceedings of the 14th Symposium on Operating Systems Principles*, pages 15–28, 1993.

[5] R. M. English and A. A. Stepanov. Loge: a self-organizing disk controller. In *Proceedings of 1992 Winter USENIX*, pages 237–251, 1992.

[6] G. Ganger and Y. Patt. Soft updates: A solution to the metadata update problem in file systems. Technical Report CSE-TR-254-05, University of Michigan Department of Electrical Engineering and Computer Science, 1995.

[7] G. R. Ganger and Y. N. Patt. Metadata update performance in file systems. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, pages 49–60, November 1994.

[8] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, San Mateo, CA, 1993.

[9] R. Hagmann. Reimplementing the Cedar file system using logging and group commit. In *Proceedings of the 11th Symposium on Operating Systems Principles*, pages 155–162, 1987.

[10] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, Aug. 1984.

[11] M. Rosenblum. *The Design and Implementation of a Log-structured File System*. PhD thesis, University of California at Berkeley, 1992. Also availbale as Technical Report UCB/CSD 92/696.

[12] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, February 1992.

[13] M. Seltzer, K. Bostic, M. K. McKusick, and C. Staelin. An implementation of a log-structured file system for UNIX. In *Proceedings of 1993 Winter USENIX*, pages 307–326, 1993.

[14] M. Seltzer, K. A. Smith, H. Balakrishnan, J. Chang, S. McMains, and V. Padmanabhan. File system logging versus clustering: A performance comparison. In *Proceedings of the 1995 USENIX Technical Conference*, pages 325, 249–264, 1995.

[15] M. I. Seltzer. *File System Performance and Transaction Support*. PhD thesis, University of California at Berkeley, 1983.

[16] M. Stonebraker. The design of the Postgres storage system. *Readings in Database Systems*, pages 610–621, 1988.

[17] A. S. Tanenbaum. *Operating Systems: Design and Implementation*. Prentice-Hall Inc., Englewood Cliffs, N.J. 07632, 1987.