# Automatic Consistency for Disk Storage

Gopalan Sivathanu, Swaminathan Sundararaman, and Erez Zadok
*Stony Brook University*

## Abstract

We make a case for ensuring semantic consistency of data at the disk-level. With the additional knowledge of pointers inside a block-based disk, we show that strong meta-data consistency semantics can be provided by the disk. Today's consistency mechanisms operate at the software-level making disks totally oblivious to the consistent state of the data. Knowledge of consistency at the disk level enables interesting functionality which cannot be provided by traditional disks. In this paper we present a disk-level consistency enforcement mechanism and evaluate it by a prototype implementation where we provide consistency transparently underneath the Linux Ext2 file system. We show that our consistency mechanism has small overheads, and the modifications required at the software-level are minimal.

## 1 Introduction

A key challenge in persistent data storage on disk is ensuring the *consistency* of data in the face of crashes. In many cases, on-disk data is unusable unless it conforms to certain software-specific invariants that define its consistency. For example, an on-disk B-Tree with dangling pointers in some of its nodes cannot be used to locate data items. Similarly, in a file system, a directory pointing to invalid or unallocated inodes constitutes a consistency violation.

Given the importance of consistency, most file systems and other software that manage on-disk storage incorporate mechanisms to ensure on-disk consistency. While some techniques involve optimistically updating on-disk state and then *fixing* consistency violations based on a disk scan (e.g., `fsck`), more modern techniques such as journalling [2] or Soft updates [1] involve constraining updates in such a way that consistency is enforced. These mechanisms are quite complex; for example, modern file systems owe a significant portion of their complexity to satisfying this requirement.

This traditional approach to managing consistency entirely at the file system or software is fraught with two key weaknesses. First, the disk system is completely oblivious to the consistency of the data it stores, which constrains the range of functionality it can provide. For example, today's block-based disk systems cannot perform consistent snapshotting of data. Snapshotting is a popular and useful feature in the storage industry, but consistent snapshotting has so far been restricted only to storage systems exporting a richer NFS-like interface [3]. Similarly, modern storage systems perform backup and asynchronous remote mirroring [5]; consistency-awareness at the storage level can increase the utility of these techniques.

A second problem with the current approach to consistency management is that every file system and every software layer that manages on-disk data is forced to duplicate the mechanisms needed to enforce consistency. This raises the bar for implementing any disk-resident data structures. Although applications can use generic transactional libraries, it often requires restructuring the application to be aware of transactions and tracking transaction context across concurrent, asynchronous operations. For example, although the journalling block device (JBD) layer in Ext3 provides a transactional interface, the Ext3 codebase had to go through a substantial amount of restructuring to actually use JBD [13] and other journalling file systems in the Linux kernel such as JFS and ReiserFS do not use JBD and have their own journalling mechanism.

To address these problems, we present *ACE-Disk*, an **A**utomatic **C**onsistency **E**nforcing Disk, a disk system that preserves the semantic consistency of stored data. In our approach, the disk system takes responsibility for consistency management, and thus is empowered to provide consistency-aware functionality such as snapshotting. Applications simply inform the disk about the relationship between various blocks that the application already knows about. Specifically, we advocate using a *Type-Safe Disk* (TSD) [10], a disk system that is aware of the pointer relationship between blocks, to get consistency, with minimal modifications at the software-level.

We present various techniques that a TSD needs to derive and enforce consistency based on pointer relationships. We show that with minimal changes at the file system level to *inform* the disk about pointers, one can get the same consistency properties that more complex techniques such as soft updates [1] provide. One of the key challenges in implementing consistency that is oblivious to the higher layers is guaranteeing periodic and timely updates to the disk state; because of inherent asynchrony in file system updates, a continuous workload at the file system could lead to the disk state being always inconsistent and thus never committed. We present techniques to bound the interval between commits despite this lim-

itation.

We illustrate the efficacy of our approach by implementing disk-level consistency for the Linux Ext2 file system and obviate the need to run fsck on Ext2 after crashes.

Overall, we find that our consistency mechanism has acceptable overheads. For a normal user workload, our modified Ext2 file system over ACE-disk introduces an overhead of just 4-5% compared to regular Ext2.

The rest of the paper is structured as follows: Section 2 discusses an extended motivation for disk-level consistency. In Section 3, we discuss some background and we present the design of our consistency mechanism in Section 4. In Section 5 we discuss our modifications to Linux Ext2 to support ACE-disks. We evaluate our mechanism and the case-studies in Section 6 and discuss some related work in Section 7.

## 2 Extended Motivation

Consistency awareness in disks enable useful functionality that cannot be provided by traditional disks. In this secion, we brief a few of those.

**Disk-level Snapshotting.** When disks can differentiate between consistent and inconsistent states of block data, they can perform efficient snapshotting or backup of a consistent state. Performing disk level backup is in most cases better than software level mechanisms in terms of performance as the disk can use its internal knowledge to perform efficient reads and writes [3, 4]. For example, a disk can use its idle time to copy data being backed up, or the current head position to interleave I/O while performing background snapshots.

**Remote Mirroring.** In enterprise storage systems, data needs to be updated asynchronously in multiple read-only mirrors that exist in geographically distributed storage systems [5]. Each such update must be semantically consistent. If the disk system knows about consistent states, it can perform updates to the mirrors during its idle time in an efficient manner.

**Preserving Trust Boundaries.** In some security infrastructures, the disk system and the software system has different trust characteristics. For example, the software layer may not be trusted and security can be enforced by the disk [10, 12]. In such systems software-level enforcement of data consistency may require relaxing security policies for an "administrative software interface". Disks tracking consistency is useful to preserve trust boundaries in this case.

## 3 Background

In this section, we discuss the background behind storage software and the importance of pointers in the context of disk storage.

### 3.1 Overview of File Systems

Several applications need to store data persistently on secondary storage disks. Storage software such as file systems and databases provide a generic interface to access storage devices and maintain their own structures to track abstractions. For example, each file system has its own on-disk layout. In this section, we provide a background of file systems in general and about the layout of the Ext2 file system in particular. We also discuss briefly a few other common storage structures that software use to manage data on disk.

File systems abstract raw disk blocks into logical entities such as file and directories. To track the set of blocks that constitute a logical file or directory, a file system uses various forms of *meta-data*; such meta-data can be broadly classified into directories, file-specific meta-data, and structures required for free-space management. Directories link logical file identifiers to file specific meta-data. File-specific meta-data contains the file attributes, and links to the actual data blocks. Allocations structures are bitmaps and free-lists required for managing disk resources such as free blocks. In common Unix file systems that follow the semantics of the Berkeley Fast File System [7], per-file meta-data objects are called *inodes*.

**Layout of the Ext2 File System.** The Ext2 file system which has its roots in Unix FFS, groups together a fixed number of sequential blocks into a block group and the file system is managed as a series of block groups. This is done to keep related blocks together. Each block group contains a copy of the super block, inode and block allocation data-structures, and the inode blocks. The inode table is a contiguous array of blocks in the block group that contain on-disk inodes. The number of inodes and their location are statically determined during the *mkfs* operation. Each inode block can contain several inodes. Each inode inside a block is treated as an *allocatable* unit, and bitmaps keeps track of allocated and free inodes within a block group.

### 3.2 Pointer Consistency in Storage

Pointers are fundamental means by which storage software organize data into logical abstractions. Today's block-based disks export a flat array-like abstraction of fixed size blocks. To manage data in the form of groups (e.g., a file) and to provide the notion of hierarchy (such as directories), they need to manage pointers between blocks. Such pointers are vital entities in storage and in most cases they impact the accessibility of data. For example, when an inode block is lost, all data pertaining to the corresponding files become unreachable and hence inaccessible. More importantly, the *consistency* of these pointers determines to a large extent the seman-

tic consistency of the information stored in a disk. For example, during a `rename` operation in Ext2, a directory entry (which is a pointer to an inode block) in a directory block is removed and added in another directory block. When the system crashes after the removal operation is done, a file becomes inaccessible even though its data items are intact. While complex storage software maintain strong forms of consistency such as the consistency between the size field in an inode and the actual file size, mere pointer consistency is sufficient in most cases. For example, if all pointers from an inode are consistent, the size field can be re-constructed by just looking at the set of pointers. In this work, we focus on ensuring pointer consistency at the disk level.

### 3.3 Pointer-Based Interface to Disk

Type-Safe Disks [10] propose an extension to the existing block-based interface (e.g., SCSI) which enables higher level software such as the file system to communicate pointers to the disk. For example, an Ext2 file system can communicate to the disk, the block pointers stored in an inode block. In addition, TSDs also manage free-space on their own thereby freeing file systems of free-space management. Block allocations are performed by the disk through an explicit ALLOC_BLOCK primitive which allocates a block from the disk-maintained free-block-list and creates a pointer to it from a *reference block*. Reference blocks are blocks with outgoing pointers and every block allocation must be made in the context of an already allocated reference block. For example, an Ext2 file system has to allocate a data block in the context of an inode block or an indirect block.

Pointers can be created and deleted through the CREATE_PTR and DELETE_PTR disk primitives respectively. TSDs enforce the constraint that all allocated blocks must have at least one incoming pointer to them (i.e., all allocated blocks must be reachable from at least one other block). When the last incoming pointer to a block is deleted, the block is automatically garbage collected by the disk. Therefore, the TSD interface does not have an explicit primitive for freeing blocks. Predetermined ROOT BLOCKS are never allocated or freed and can be used to bootstrap block allocation.

Unlike traditional disks, TSDs have complete information about the relationships between blocks and hence TSDs can differentiate live blocks from dead blocks and reference blocks from regular data blocks.

## 4 Design

Our disk-level consistency mechanism enforces the following constraint: the on-disk version of data should always be consistent. To accomplish this, we need to discover semantically consistent groups of blocks and commit them atomically to the disk when they are written by higher level software such as the file system. All inconsistent block updates should be buffered inside the disk until they become consistent. For example, when a new file is created, the corresponding directory block and the inode block have to be updated. When just one of the writes arrives at the disk it indicates an inconsistent update. In that case, we need to buffer the update until the second block write also arrives. When both the directory block and inode block writes have arrived at the disk, we need to ensure (at the disk level) that both these blocks are committed atomically to stable storage.

We describe how update dependencies between blocks can be inferred from pointers in Section 4.1. In Section 4.2 we present our enhanced pointer interface that make dependency inference robust. We describe the consistency enforcement process in Section 4.3 and discuss a key issue in disk-level consistency enforcement in 4.4. We finally detail our prototype implementation of the system in Section 4.5.

### 4.1 Inferring Dependencies from Pointers

Determining semantic relationships between blocks at the disk level requires additional information exchange between the software layer and the disk. Today's block-based disks treat all stored information as opaque data and they do not have knowledge of data semantics. For example, today's disks cannot differentiate between a data and meta-data block in a file system. We leverage the idea of Type-Safe Disks (TSDs) [10], to obtain pointer-relationships between blocks as maintained by the higher level software.

Pointers at the disk level not only convey structural information about data items stored on disk, but also they enable the disk to infer dynamic relationships between blocks that get updated. For example, a when a new block $a$ is allocated and a pointer is created to it from another block $b$, both $a$ and $b$ depend on each other. If the system crashes when just one of the blocks is updated, the disk is left in an inconsistent state. This is because, if only block $a$ is updated, it would be pointing to a block with junk data (not yet written), and if only $b$ is updated, it becomes unreachable as there would be no incoming pointers to it.

The existing TSD interface consists of primitives for allocation and pointer operations, as described in Section 3. We discuss how each TSD primitive can be used to infer update dependencies.

**Allocation.** The allocation primitive internally creates a pointer to the newly allocated block, in the reference block passed. This operation relates two blocks: the newly allocated block and the reference block. Updating one of the blocks alone clearly leaves the system in an inconsistent state; hence these two blocks constitute

a dependency constraint and they have to be committed atomically to stable storage.

**Pointer Creation.** The pointer creation primitive creates a pointer from any two arbitrary allocated blocks. In this case, the source block *must* be written subsequent to the pointer creation operation to write the new pointer value in it. However, the destination block need not necessarily be written, as the it is a previously allocated block. For example, while creating a new file in the Ext2 file system, a pointer gets created from the directory block to an already allocated inode block that contains the inode of the new file. In this case, both these blocks constitute a dependency. This is because the directory block has to be updated with the new pointer to the inode block, and the inode block has to be updated with valid information about the newly created file. Failure to commit the latter will result in a directory entry pointing to an invalid inode. As a counter example, if we consider a common index-based storage structure, a set of index blocks point to data block. In this case, duplicating an index block for reliability reasons would result in creation of new pointers from the duplicated index block to the existing data blocks. Here only the index block needs to be written and not the data blocks. Therefore, the pointer creation primitive provided by TSD does not convey enough information to decide whether or not the source and destination blocks constitute a dependency.

**Pointer Deletion.** A pointer deletion operation deletes an existing pointer from block $a$ to block $b$. This operation has a special case: if the deleted pointer is the last incoming pointer to block $b$, we garbage collect $b$ and it can be re-allocated during future allocation requests. In both cases, it is clear that block $a$ has to be written subsequent to this operation for it to reflect the pointer deletion. The destination block $b$ in the case of garbage collection need not be written. However, it does constitute a dependency: $b$ must not be re-allocated until $a$ is written. For example, when the last pointer from an inode block to a data block is deleted during a `truncate` operation, re-allocating the data block to another inode before the old inode is written could result in a state where the old inode points to the contents of a different file. In the normal case of a pointer deletion where garbage collection does not occur, we cannot infer whether the source and destination constitute a dependency for the same reason as explained in the case of pointer creation.

## 4.2 An Enhanced Pointer Interface

As described in the previous section, the pointer API exported by a TSD do not always convey enough information to make correct inferences in a generic manner. In this work, we fine-tune the TSD API to make it more complete in terms of conveying pointer information.

We introduce the notion of a *sub-block* in a TSD. We use sub-blocks to formalize *allocatable* units inside a block, as maintained by the higher-level software. For example, in Ext2 each inode block can contain several inodes, each of them allocated and freed at the software level. Although formalizing these units in a precise manner requires knowledge about the unit size and offsets inside a block, we just need a rudimentary knowledge of sub-blocks to infer dependencies. For example, to decide whether or not a create or delete pointer operation constitutes a dependency we just need to know if that pointer points to a sub-block. This intuition is based on the fact that, to preserve pointer consistency we need to guarantee two properties: first, no pointer points to unwritten (junk) units, and second, no allocated units become unreachable. In our inference mechanism we make use of additional disk primitives for creating and deleting pointers to sub-blocks. Note that the disk need not track information about sub-blocks, but it just needs to dynamically know sub-block pointer operations by way of explicit primitives. Higher-level software call the respective sub-block primitives while creating and deleting pointers to newly allocated or freed sub-blocks. For example, Ext2 has to call a sub-block pointer creation primitive to create a pointer between a directory block and inode block while creating a file. From this we can infer that the directory and inode blocks form a dependency constraint. The complete specific of the extended TSD API is mentioned in Section 4.2.

We present an extended pointer interface to TSDs that captures most cases of dependency inferences. In the primitives described below, the parameter $t$ refers to a logical timestamp value for the operation. This is to let the disk know about the temporal ordering of operations as they are issued by the higher level software. The purpose and usage of this parameter is discussed in detail in Section 4.3.

1. READ($Blockno$): Block read primitive.
2. WRITE($Blockno$, $t$): Block write primitive.
3. ALLOC_BLOCK($Ref$, $t$): Allocates a new block $a$ from the disk-maintained free-block list and creates a pointer to it in $Ref$. Both $Ref$ and $a$ constitute a write dependency constraint.
4. CREATE_PTR($Src$, $Dest$, $t$): Creates a new pointer from $Src$ to $Dest$. This primitive does not create any dependency.
5. DELETE_PTR($Src$, $Dest$, $t$): Deletes an existing pointer from $Src$ to $Dest$. If this is the last incoming pointer to $Dest$, $Dest$ is garbage collected (marked free) and it creates a new dependency between the write of $Src$ and the re-allocation of $Dest$.
6. MOVE_PTR($Src$, $Dest$, $Newsrc$, $t$): Moves the source block of an existing pointer from $Src$ to

$Newsrc$. This operation results in creation of a new dependency for the writes of $Src$ and $Newsrc$. This primitive is useful for handle cases such as a `rename` operation in a file system, or a B-tree node split where pointers need to be moved from one block to another.

7. ALLOC_SUB_BLOCK($Ref$, $Target$, $t$): Creates a new pointer between block $Ref$ and block $Target$. $Target$ is a block that contains multiple allocatable software-level structures as described in Section 4.1. This primitive is called when a software-level structure in $Target$ is allocated. This disk does not track these structures. This creates a new write dependency between $Ref$ and $Target$. The disk differentiates this primitive from the CREATE_PTR primitive only to infer dependencies.
8. FREE_SUB_BLOCK_PTR($Ref$, $Target$, $t$): Deletes an existing pointer between $Ref$ and $Target$. $Target$ is a block that contains multiple allocatable software-level structures. This primitive is called when a software-level structure in $Target$ is freed. If this operation deletes the last incoming pointer to block $Target$, $Target$ is garbage collected and a new dependency is created between $Ref$ update and re-allocation of $Target$. If the pointer deleted is not the last incoming pointer to $Target$, a new dependency is created for the update of $Ref$ and $Target$.

## 4.3 Consistency Enforcement

In this Section we detail how an ACE-disk guarantees consistent data commits to stable storage. Figure 1 shows the overall architecture of an ACE-disk.

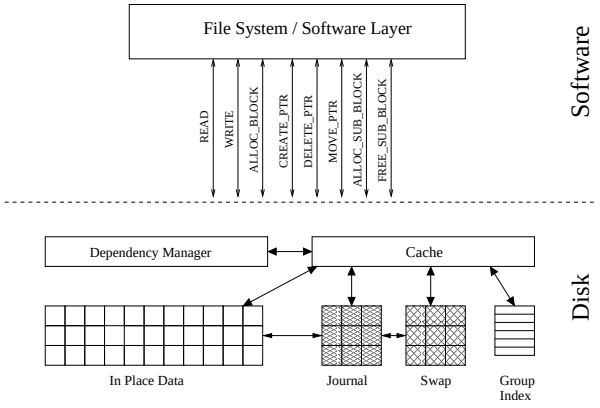


*Figure 1: Architecture of an ACE-disk*

An ACE-disk consists of five main components: (1) *dependency buffer*, a buffer layer made of high-speed memory where inconsistent block updates are buffered until the corresponding dependency becomes consistent; (2)*buffer swap space*, a swap area in the disk which is used to swap out inconsistent buffer data when the cache is full; (3) *journal space*, an area on disk which is used to ensure atomic update of resolved dependencies; (4) *group manager*, which tracks the pointer operations and constructs dependencies; *group index* a data-structure used by the group manager to store disjoint dependencies and the blocks affected by each of those dependencies. The buffer layer acts both as a read and write cache, and gets invalidated during power down of the disk. All inconsistent block updates are buffered in the cache to ensure that the state of data stored in place is always consistent. The swap space is used when the number of inconsistent blocks exceed the size of the high speed buffer memory.

When an ACE-disk infers a dependency during a pointer operation, it associates a *group* object with that dependency. This group object contains information about the set of blocks that are affected by that dependency. We use the terms *group object* and *dependency group* interchangeably in the rest of the paper to refer to a list of blocks that needs to be committed atomically to stable storage to ensure consistency. A *group entry* refers to a member of a group which contains a block number and the time at which it was added. When a block is written *after* it is added to a dependency group, the corresponding group entry for that block is marked "ready." When all entries in a dependency group are ready, the group is said to be *resolved*, and all blocks associated with it can be committed atomically to the disk.

In a simple case, when the first pointer operation happens in a disk causing a dependency creation between two blocks $a$ and $b$, a new dependency group $G$ is created and both the blocks are added to it. When write requests for both $a$ and $b$ have arrived at the disk, the dependency group $G$ is said to be *resolved* and all the blocks in $G$ can be committed atomically to the disk. However, if another pointer operation happens before $G$ is resolved introducing a dependency between blocks $b$ and $c$, the operation *extends* the existing dependency group. This is because, one of the blocks in the new dependency (block $b$) is already part of an existing dependency. Thus, in this scenario block $c$ should be added to group $G$ as well. Therefore, whenever there is a new dependency introduced between any two blocks $x$ and $y$ by way of a pointer operation, one of the following three actions are taken:

1. If both $x$ and $y$ are not part of any existing dependencies, a new dependency group is created and $x$ and $y$ are added to it.
2. If only one of $x$ or $y$ is associated with an existing dependency group $G$, then both blocks are associated with $G$ and are marked "not ready."
3. If both $x$ and $y$ are already associated with the

same group $G$, then no group action needs to be taken. However, the entries in the group pertaining to blocks $x$ and $y$ have to be marked "not ready" as a new constraint is added between the two blocks.

4. If both $x$ and $y$ are associated with different groups $G_1$ and $G_2$, then $G_1$ and $G_2$ are *merged*, and the entries for $x$ and $y$ are marked "not ready"

As pointer operations construct dependencies between blocks, higher-level software must ensure that the pointer management primitives are issued to the disk *before* the source and destination blocks are updated. This constraint is implicitly enforced for the block allocation primitive as a block cannot be updated before it is allocated. However for the pointer creation and deletion primitives, higher-level software has to ensure that it follows this ordering rule. For example, when a `create` happens in Ext2, the sub-block pointer creation primitive has to be issued for the directory and the inode blocks before the contents of the blocks are updated.

**Temporal Ordering of Operations.** ACE-disk's consistency mechanism relies on the temporal relationships between operations seen at the disk level. For example, an entry in a dependency group is marked ready when a write arrives after the dependency creation. However, in today's modern operating systems and disks, operations can be re-ordered at any level. For example, file systems today predominantly perform asynchronous I/O where block writes are buffered at the software level and are flushed to the disk in regular intervals of time. Moreover, modern disk device drivers re-order or merge disk requests before issuing to the disk for performance reasons. These factors make the temporal ordering of operations that the disk sees completely different from the order that the higher-level software issued. Therefore, unless additional ordering information is communicated from the software-level, the disk cannot obtain the precise temporal order of operations.

ACE-disk solves this problem by introducing two constraints on the operations: (a) all pointer primitives take place synchronously and (b) all operations have associated logical timestamps. These two constraints enable the disk to obtain precise temporal ordering of the operations. Although synchronous pointer operations may affect performance, it is mitigated by the fact that these operations do not result in block I/O inside the disk, in the critical path. Timestamps in this case are logical. For example they can be a monotonically increasing sequence number. Whenever higher-level software issues a pointer operation, it has to pass a sequence number along with it. Similarly when the in-memory copy of a disk block is updated by the software, a sequence number has to be associated with the buffer for that block. Whenever a pointer operation introduces a dependency, its sequence number is associated with the corresponding group entries. The entries are marked ready only when a subsequent write arrives with sequence number greater than the stored one. Note that introducing sequence numbers with block I/O operations is simple—we have modified the Linux kernel to support sequence numbers along with buffers whenever they are dirtied. This modification was trivial and required changing just 50 lines of code.

**Dependency Commits and Crash Recovery.** When a dependency group is resolved all blocks in the group has to be committed in place atomically. A power failure while committing a dependency group should not leave the in place data in an inconsistent state. ACE-disk uses a logging mechanism to ensure this. All blocks in a resolved groups are first written to a log and synced with a commit identifier before the in place commit happens. The log is discarded when the in place commit is complete. After a crash, an ACE-disk checks the log for valid group data and replays them. The log contains separate journals for each dependency group and hence each of them are replayed after the crash to bring the system to a consistent state.

## 4.4 Bounding Commit Interval

The amount of data lost during a crash depends on the interval between the instant a block write arrives at the disk and the time when it is actually committed to stable storage. In an ACE-disk, inconsistent block data gets buffered until the entire dependency group is resolved. ACE-disk's mechanism of managing dependency groups allow extending a group whenever pointer operations happen from or to a member of the group. Thus, during normal operation, a dependency group could potentially get extended repeatedly during a continuous workload that performs pointer operations. For example, in Ext2, for a recursive directory creation workload, the entire working set would form part of the same dependency group as all blocks branch out from the inode of the root directory. Moreover, as pointer operations always precede the block write operations, a dependency group could never get resolved for a continuous workload. This is because before the time when all blocks in a group are marked ready, the group could be extended several times with new blocks or new dependencies for the existing blocks. This results in two problems. First, large amounts of data may get lost in the event of a crash, although the on-disk state is consistent. Second, excessively long dependency groups require buffering of a large number of blocks and hence impose onerous space requirements.

Bounding the interval between dependency commits is challenging particularly at the disk level because the disk has no knowledge about intermediate versions of

block data that are known to the higher-level software. This is because most higher-level software buffer writes and hence the versions of block data that reach the disk could be a small subset of total number of versions that the software knows about. For example, if a file is created in Ext2, an inode block is modified. Before the inode block write is issued to the disk, if another file is created whose inode is in the same block, the disk sees only the version of the block updated with both inodes. Therefore, the disk cannot spawn a new dependency group during a pointer operation for a block, when the existing group containing a block has reached a time threshold.

Blocking pointer operations at the disk level until an existing dependency is committed could be a solution to the bounding problem, but requires radical modifications to the higher-level software to support it. This is because software such as file systems perform locking of data-structures at an operation level. When a pointer operation blocks, the file system could sleep after grabbing a lock on the data-structure which reside on a block that needs to be committed for some dependency to resolve. This could result in a deadlock as the block containing the data-structure cannot be committed until the operation in execution completes.

An ACE disk solves this problem by having new error modes for pointer creation operations. The allocation and pointer management primitives could optionally return one of the following errors to the higher-level software: SYNC_BOTH, SYNC_SRC, or SYNC_DEST. As the names indicate, the disk can fail a pointer operation and choose to request the higher level software to write the source, destination, or both blocks associated with that operation. Upon receiving one of these errors the software should issue a write of the current version of the corresponding blocks, and then retry the pointer operation. At the disk level, whenever a dependency group is unresolved beyond a time threshold it is *frozen*. Whenever new dependencies are created for a block that is already part of a frozen group and in an "not ready" state, the disk returns one of three errors mentioned above, depending on whether the block is the source, destination, or when both the source and destination blocks exist in frozen groups in "not ready" state. This way of forcing the software to commit the intermediate version of the data helps the disk to spawn new dependency groups for blocks that are already ready in a frozen group. An ACE-disk ensures that at a block is never part of more than two groups at a time, the older of which is frozen. This is done by ensuring that a group is not frozen until all blocks in the group are not part of any other frozen group. This method ensures commit of dependency groups in tune with the block write interval of the higher level software. We verified the correctness of our bounding solution by implementing this in the Ext2 file system. Each every case, the commit interval of the dependency groups were in tune with that of the software level write-back interval.

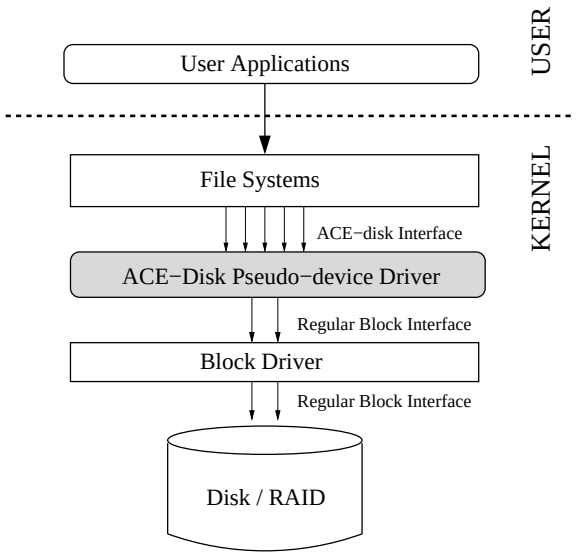


*Figure 2: Implementation Architecture*

## 4.5 Implementation

We implemented a prototype ACE-disk as a pseudo-device driver in the Linux kernel 2.6.15 that stacks on top of an existing disk block driver. The pseudo device driver layer receives all block requests, and redirects the common read and write requests to the lower level device driver after the required processing. The additional primitives required for operations such as block allocation and pointer management are implemented as driver `ioctl`s. The architecture of our implementation framework is shown in Figure 2

To enable sequence numbers with block I/O requests, we added a new field to the buffer header object and the `request` token object in the Linux kernel. Whenever a buffer is marked dirty, we generate a sequence number and update it in the buffer header. When a write is issued for a buffer, the sequence number is carried over to the `request` object and hence available to the ACE-disk pseudo-device driver. Sequence numbers are generated by an atomic increment of a counter value. The same counter value is used during pointer operations and modifying buffers.

Our prototype ACE-disk contained 6900 lines of kernel code of which 3060 lines of code were reused from the existing TSD prototype.

### 4.6 Limitations of Pointer-driven Consistency

While the update dependency information conveyed by pointers is quite rich and as we show, sufficient to enforce consistency, it has some limitations when compared to the more general notion of transactional consistency. Specifically, the dependency information conveyed by pointers is limited to a pair of blocks; e.g. if a pointer is created between two blocks, the two blocks will be updated atomically. However, our mechanism cannot support atomic commits of an arbitrary group of blocks. For example, on creation of a new directory ( i.e. mkdir) in ext2, a pointer is created from the parent directory block to the inode of the child directory, and the inode initialized. Then a new block is allocated for the child directory and a pointer created between the child inode and the child directory's new data block. With a transactional system, these three blocks will be committed atomically. But in our case, the first pointer creation and the initialized inode could be committed before the second pointer creation. As a result, a directory inode may end up with a state where it has no blocks at all, which is an apparent violation of consistency.

However, we argue that this consistency problem falls under a class of *online-patchable* consistency violations. For example, just by looking at the initialized directory inode with no pointers, it is unambiguous that a crash happened just before the new directory's block got allocated, so it's safe to immediately allocate a new block for the directory and assign it to the inode. Note that in contrast, a more "real" consistency problem would be a directory pointing to the wrong inode, perhaps a regular file inode, where it is not obvious what the correct state should be. Pointer consistency could lead to such transient online-patchable consistency violations the violation is readily and unambiguously identifiable and the fix for that is obvious as well. Most importantly, the fix to such a violation is *local*, in that it does not require looking at the global state of the file system. We believe that the pointer-derived consistency semantics is thus a useful and simpler counterpart to the more general transactional consistency.

## 5 Case Study: Ext2ACE

We modified the Ext2TSD file system [10] to support ACE-disks. Ext2TSD is a modified version of Ext2 that support the Type-Safe Disks interface. Ext2TSD does not manage free-space on its own; instead it uses the allocation and delete pointer primitives of TSDs to allocate and free data. It also uses disk primitives for creating and deleting pointers between directory blocks, inode blocks, indirect blocks, and data blocks. Ext2TSD already conforms to the constraint that pointer operations have to be issued before modifying the contents of the corresponding source and destination blocks.

To modify Ext2TSD to Ext2ACE, we had to change the following:

**Use sequence numbers.** We modified Ext2TSD to associate sequence numbers with every pointer operation and block write I/O. We included a new atomic counter value in the in-memory super block of Ext2 and use this counter as the sequence number. Whenever a buffer is marked dirty by the file system (inodes, directories, indirect blocks), the timestamp field of the buffer is updated with the new sequence number.

**Handle new error modes.** We handled the error modes returned by ACE-disk during pointer operations (described in Section 4.4). The block allocation and pointer management calls in Ext2TSD are modified to handle sync requests from the disk. When a pointer operation results in an error, the file system writes the current version of the corresponding blocks to the disk, and retries the pointer operation.

**Handle rename dependency.** We used the MOVE_POINTER disk primitive during `rename` operation. This is because a `rename` of a file could delete an entry from a directory block $old$ and create it in another directory block $new$. The pointer between $new$ and inode block of the file to be renamed has to be updated to point from block $new$ instead of $old$. Note that blocks $old$ and $new$ has to be updated atomically as a crash while just one of them is updated could result in loss of the file. The MOVE_POINTER primitive creates a new dependency between the two source blocks for the move.

**Change inode update mechanism.** The Ext2 file system adopts a special policy to update inode blocks. An Ext2 inode block can contain several inodes and each inode is treated as an allocatable unit. Whenever an inode needs to be written to the disk, the contents of that particular inode is alone updated and not the other inodes in that block, even though the in-memory copy of the other inodes of the other inodes are dirty. This update mechanism confuses an ACE-disk as it operates at the granularity of an entire block. When a block write arrives at the disk, an ACE-disk assumes that the content of that write reflects the entire state of the in-memory copy of that block when it was modified. Therefore this update mechanism can make an ACE-disk to erroneously resolve dependencies. We therefore modified the inode update mechanism of Ext2 such that it commits the in-memory state of all inodes in a given inode block when it is written to the disk.

**Use sub-block primitives.** We used the ALLOC_SUB_BLOCK and FREE_SUB_BLOCK primitives

while creating and deleting pointers between the directory blocks and inode blocks. Therefore, new dependencies get created between the parent directory block and the child inode block during the CREATE, MKDIR, UNLINK and RMDIR operations.

Overall, the amount of changes required to port Ext2TSD to Ext2ACE was minimal. We just added 530 lines of new code and modified 160 lines of existing code.

# 6 Evaluation

We evaluated the performance of our prototype ACE-disk using Ext2ACE. We ran both a general purpose workload and a micro-benchmarks on our implementation and compared it with a regular Ext2 and Ext3 file systems running on a normal disk. We compared our system with Ext3 because it is a journalling file system that provides similar consistency guarantees as ACE-disk at the software level. For all benchmarks we used Ext3 in its default journalling mode (ordered writes mode). In this mode file meta-data alone is journalled and it is written to the journal only after the corresponding data blocks are written directly in place.

As our prototype ACE-disk is implemented as a software layer, it uses the CPU and memory of the host system which are shared by the file system and other software. In a real environment, the portion of code that exists in our pseudo-device driver would go inside the disk, and hence it will be using isolated resources associated with the disk. Therefore, the CPU overheads that we see in our benchmark results are not fully representative of the overheads in a real environment. Similarly, the pointer management disk primitives are implemented as software level `ioctl`s and hence factors like bus latency and other hardware-related issues are not shown by our benchmark results.

We conducted all tests on a 2.8GHz Xeon with 1GB RAM, and a 74GB, 10Krpm, Ultra-320 SCSI disk. We used Fedora Core 4, running a vanilla Linux 2.6.15 kernel. To ensure a cold cache, we unmounted all involved file systems between each test. We ran all tests at least five times and computed 95% confidence intervals for the mean elapsed, system, user, and wait times using the Student-$t$ distribution. In each case, the half-widths of the intervals were less than 5% of the mean. Wait time is the elapsed time less CPU time used and consists mostly of I/O, but process scheduling can also affect it.

For all benchmarks we included the file system unmount time in our calculation. This is because ACE-disk commits dependency groups asynchronously using separate kernel threads, and a file system unmount procedure blocks until all outstanding threads have completed their commit operation. This is relevant even for normal Ext2 and Ext3 as they commit all outstanding dirty data during an unmount.

For an I/O-intensive workload, we used Postmark [14], a popular file system benchmarking tool. We compiled OpenSSH version 4.5 to generate a relatively CPU-intensive workload. To isolate the overheads of individual meta-data operations in a file system, we tested the `create` and `unlink` operations. In most of the benchmark results, the increased CPU time is caused by the dependency manager which tracks dependencies for every block-write I/O and for the pointer operations.

## 6.1 Postmark Results

Postmark stresses the file system by performing a series of file system operations such as directory lookups, creations, and deletions on small files. A large number of small files is common in electronic mail and news servers where multiple users are randomly modifying small files. We configured Postmark to create 30,000 files whose sizes ranging from 512 bytes to 10 KB, and perform 250,000 operations in 200 directories. This workload particularly stresses the ACE-disk as a large number of dependencies get created and resolved during the meta-data operations. The time taken for the Postmark benchmark for Ext2, Ext3, and Ext2ACE are shown in Figure 3.

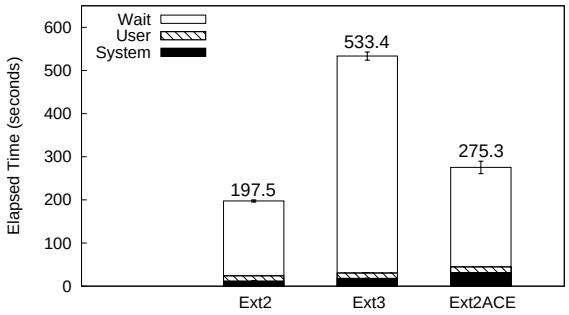


*Figure 3: Postmark Results*

Ext2ACE on top of ACE-disk had an elapsed time overhead of 40% compared to regular Ext2 on a normal disk. Although the system time increase is 2.6 times relatively, this has not contributed much to the elapsed time overhead. As mentioned earlier, this overhead is because of dependency tracking during every block write and pointer operations. The wait time increase (32%) is predominantly because all blocks are written out twice in the case of an ACE-disk to ensure atomic commits of dependency groups. All block data is written out to the journal first and after the journal is synced, in-place commits happen. Ext3 ran almost twice as slow as Ext2 because of its ordered journalling mode. Ext2ACE is faster than Ext3 in this case because ACE-disk journals both data and meta-data blocks and for a small file work-

load such as Postmark, random writes get converted to sequential ones. The in-place commit of data in ACE-disk happens in an asynchronous manner.

## 6.2 OpenSSH Compile Results

To simulate a relatively CPU-intensive user workload, we compiled the OpenSSH source code. We used OpenSSH version 4.5, and analyzed the overheads of Ext3 and Ext2ACE for the `untar`, `configure`, and `make` stages combined. These operations in combination constitute a significant amount of CPU and I/O operations. The results for OpenSSH compilation is shown in Figure 4.

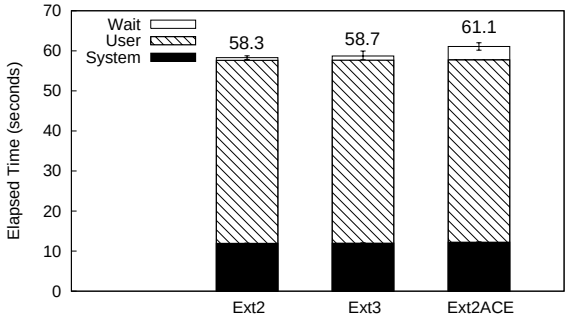


*Figure 4: Openssh Compile Results*

The times taken by Ext2 and Ext3 for the compilation workload are almost similar. This is because this is a mostly CPU-intensive workload. Ext2ACE had an elapsed time overhead of 5% compared to Ext2 and Ext3. This is because of the increase in wait time (1 sec vs. 3.4 secs). The increase in wait time is caused by the CPU context switches between the main compilation process and the asynchronous dependency commit threads of ACE-disk. Since this is a CPU-intensive workload, the context switch time is more pronounced than Postmark. In a real environment, as the dependency commits are performed inside the disk, this context switch overhead would not be seen. The system time overhead is not significant for Ext2ACE in this case because there are relatively few I/O operations that require processing to track dependencies.

## 6.3 Micro-Benchmarks

We ran two micro-benchmarks to obtain the overheads of the `create` and `unlink` file system operations. We evaluated these two operations because both of them exercise the ACE-disk's dependency trackers and consistency enforcement mechanism. For the create workload, we created 500 directories with 1,000 files each totaling to 500,000 files. For the unlink workload, we removed all created files and directories. The results of the `create` and `unlink` workloads are shown in Figures 5 and 6, respectively.

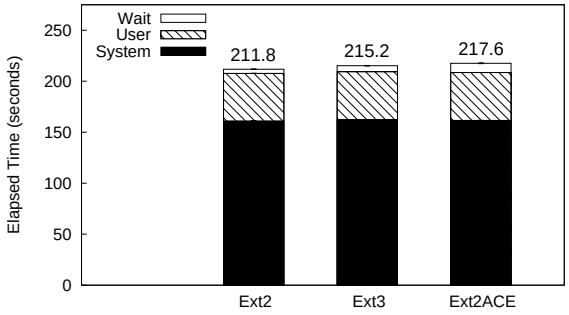


*Figure 5: Create Micro-Benchmark Results*

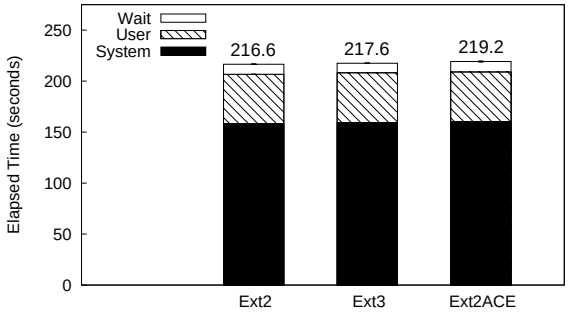


*Figure 6: Unlink Micro-Benchmark Results*

For the `create` workload, Ext2ACE had an overhead 2.7% compared to Ext2. This is mostly caused by the increase in wait time due to the additional I/O operations writing out block data twice for ensuring atomicity in block commits. For the `unlink` workload the results of Ext2ACE is similar to Ext2 and Ext3 as `unlink` results in smaller number of writes than creates, because freed blocks are not written to the disk.

Overall ACE-disks have small overheads for normal user workloads. When the workload is highly I/O-intensive, more information needs to be tracked by the disk to manage dependencies. This results in more CPU time which is mitigated by the fact that the disk uses its own isolated CPU in a real environment.

# 7 Related Work

Consistency mechanisms for file systems have been explored extensively. Early file systems such as FFS [7] relied on a global scan of disk metadata to fix consistency problems. This mechanism, called the file system consistency check (fsck) was in popular use until recently in the Linux Ext2 and Windows VFAT file systems. However, as increasing disk sizes made such global scans more and more expensive, more efficient mechanisms have become popular. Journalling, originally proposed

as early as in the Cedar file system [2], uses database like transactions for metadata updates. Modern file systems such as Ext3 and Windows NTFS use journalling for file system consistency. Another technique proposed for file system consistency is Soft Updates [1, 6], which orders updates carefully so that pointer dependencies get updated in the right order. Soft updates is somewhat similar in spirit to our approach since it is also pointer-based. A relatively recent study evaluated the trade-offs between journalling and soft updates [9].

Database systems have for long used mechanisms for consistency. Consistency in databases is enforced via transactions; the ARIES transaction based recovery mechanism [8] is used quite widely in database systems. The basic technique is to group all related updates into a single transaction that is then committed to disk atomically, so that the state remains consistent. As we described in Section 4.6, transactions are more general and powerful than pointer-based consistency, but using transactions requires a fair bit of work at the application level. Our mechanism provides a simpler yet effective alternative to transactions, although not as general.

Implementing consistency at the disk level transparent to the file system has been explored in the context of Semantically-smart disks (SDS) [11]. In that paper, the authors implement journalling underneath unmodified Ext2 by utilizing inferred semantic knowledge. However, in their work, the disk system had to be aware of the specific structures at the file system level and thus was tied to a specific file system. Further, it required a synchronous mount of the file system. Our work explores enforcing consistency in a manner generic to the higher level software. However, in the process, we require changing the file system or software above to use the pointer API, as compared to SDS which did not require any change. We therefore view both these approaches as complementary.

## 8 Conclusions

In this work, we have shown how pointer knowledge at the disk-level can enable inference and enforcement of semantic consistency of data that gets written. As pointers are fundamental entities in disk storage, our mechanism is generic across several storage applications. Disk-level knowledge of consistency also enable interesting applications as such on-disk snapshotting. Our prototype implementation of Ext2ACE shows that it is simple to modify storage software to support ACE-disks. Evaluation of our implementations show that the overheads associated with our consistency mechanism are minimal and are comparable to existing software-level mechanisms.

While the update dependency information conveyed by pointers is quite rich and sufficient to enforce consistency, they do not provide the strong and flexible consistency guarantees provide by transactional systems. However, for most common storage applications the pointer-based consistency mechanism would be adequate.

## References


[1] G. R. Ganger, M. Kirk McKusick, C. A. N. Soules, and Y. N. Patt. Soft updates: a solution to the metadata update problem in file systems. *ACM Trans. Comput. Syst.*, 18(2):127–153, 2000.

[2] D. K. Gifford, R. M. Needham, and M. D. Schroeder. The Cedar File System. *Communications of the ACM*, 31(3):288–298, 1988.

[3] D. Hitz, J. Lau, and M. Malcolm. File System Design for an NFS File Server Appliance. In *Proceedings of the USENIX Winter Technical Conference*, pages 235–245, San Francisco, CA, January 1994.

[4] N. C. Hutchinson, S. Manley, M. Federwisch, G. Harris, D. Hitz, S. Kleiman, and S. O'Malley. Logical vs. Physical File System Backup. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI 1999)*, pages 239–249, New Orleans, LA, February 1999. ACM SIGOPS.

[5] M. Ji, A. Veitch, and J. Wilkes. Seneca: remote mirroring done write. In *Proceedings of the Annual USENIX Technical Conference*, San Antonio, TX, June 2003. USENIX Association.

[6] M. K. McKusick and G. R. Ganger. Soft Updates: A Technique for Eliminating Most Synchronous Writes in the Fast Filesystem. In *Proceedings of the Annual USENIX Technical Conference, FREENIX Track*, pages 1–18, Monterey, CA, JUNE 1999. USENIX Association.

[7] 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, August 1984.

[8] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17(1):94–162, 1992.

[9] M. I. Seltzer, G. R. Ganger, M. K. McKusick, K. A. Smith, C. A. N. Soules, and C. A. Stein. Journaling Versus Soft Updates: Asynchronous Meta-data Protection in File Systems. In *Proc. of the Annual USENIX Technical Conference*, pages 71–84, San Diego, CA, June 2000. USENIX Association.

[10] G. Sivathanu, S. Sundararaman, and E. Zadok. Type-Safe Disks. In *Proceedings of the 7th Symposium on Operating Systems Design and Imple-*

*mentation (OSDI 2006)*, pages 15–28, Seattle, WA, November 2006. ACM SIGOPS.

[11] M. Sivathanu, V. Prabhakaran, F. I. Popovici, T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Semantically-Smart Disk Systems. In *Proceedings of the Second USENIX Conference on File and Storage Technologies (FAST '03)*, pages 73–88, San Francisco, CA, March 2003. USENIX Association.

[12] J. D. Strunk, G. R. Goodson, M. L. Scheinholtz, C. A. N. Soules, and G. R. Ganger. Self-Securing Storage: Protecting Data in Compromised Systems. In *Proceedings of the 4th Usenix Symposium on Operating System Design and Implementation (OSDI '00)*, pages 165–180, San Diego, CA, October 2000. USENIX Association.

[13] S. Tweedie. Journaling the Linux ext2fs filesystem. In *LinuxExpo Conference Proceedings*, May 1998.

[14] VERITAS Software. VERITAS File Server Edition Performance Brief: A PostMark 1.11 Benchmark Comparison. Technical report, Veritas Software Corporation, June 1999. `http://eval.veritas.com/webfiles/docs/fsedition-postmark.pdf`.