HEWLETT PACKARD

Mime: a high performance parallel storage device with strong recovery guarantees

Chia Chao, Robert English, David Jacobson, Alexander Stepanov, and John Wilkes

HPL–CSP–92–9 rev 1
18 March 1992, revised 6 November 1992

You must love Mime …
—Richard Wagner, *Siegfried, Scene I*

Hewlett-Packard Laboratories
Concurrent Systems Project

Mime is a shadow-writing disk storage architecture that uses late binding of data to disk locations to improve performance. This results in a simple design with strong atomicity guarantees and a rich set of transaction-like capabilities.The shadowing system in Mime is designed to avoid synchronous updates of metadata during normal operation. Instead, metadata is recovered using an efficient combination of logging and inverted indices. Mime can be used to transparently improve the performance and failure characteristics of existing file systems. The extensions that it makes to typical storage semantics, together with its high performance offer ways to simplify file system implementations.

Keywords: disk array architecture, shadowing, transactions, intelligent disk
The previous revision was also published as HP Laboratories technical report HPL–92–44.

1 Introduction

Mime[1] is a high-performance storage architecture for a disk subsystem. The Mime architecture uses shadowing, logging and checkpointing into the disk subsystem to provide a rich set recovery properties. As well as improving the reliability of updates and the semantics provided across power failures, the performance is significantly better than for a regular disk drive: for example, our simulations show that write performance for an unmodified 4.2BSD-based file system can improve by more than a factor of 2.

The primary contributions of Mime are:

- Improved performance:
  - low-latency short writes
  - even lower latency with multiple disks, coupled with improved throughput
- Improved recovery semantics for writes:
  - atomic multi-sector writes
  - separate host controls for ordering and forcing updates to disk
  - atomic group-commit of multiple writes
  - “tentative” writes that can be undone
  - faster recover after failure than previous work in this area
- Atomic snapshots of the state of the disk subsystem (e.g., for online backup)

We can summarize the Mime approach in three aphorisms:

“Update in place is a poison apple” — *Jim Gray*

“One can solve any computer science problem with an extra level of indirection” — *ancient proverb*

“Keep an extra copy of your metadata with the data” — *Butler Lampson*

The rest of this paper is organized as follows. We begin with an overview of the kind of recovery properties desirable for a storage system and follow this with a description of related work—one part of which is a key foundation for the Mime architecture. Next, we introduce the functionality and architecture of Mime itself at high level, and follow that with a description of the components of the Mime architecture. We analyze the performance impact of Mime on both existing file systems and new ones that exploit the new functionality, and conclude with a summary of results, and current status.

1.1 Recovery guarantees

Mime builds on the understanding of different recovery properties (i.e., serializability and monotonicity) in the database and file system communities. There are a number of possible consistency criteria that one could envision for a storage system; we discuss some of them here as background material for an analysis of Mime’s functionality.

The weakest guarantee is *write enqueueing*, as offered used by the 4.2BSD Fast File System [Leffler89],[2] which ensures that blocks will be written within some bounded time, such as 30 seconds, but says

1. Mime (pronounced MEE-may) was a dwarf who made a magic helmet that allowed its owner to change his shape at will.
2. We refer to this as the weakest of guarantees because anything weaker guarantees nothing at all.

nothing about ordering of writes or about the atomicity of operations or groups of operations. Additional guarantees can be obtained through the use of `sync` or `fsync` system calls, but only with a significant performance penalty. The file system does ensure some consistency within its own data structures, but only by using ordering facilities not available to application code.

A stronger guarantee, *write atomicity*, ensures that either all the data written in a single operation will appear on disk, or none of it will. While some disk drives make this guarantee at the sector level, writes are normally several sectors long, so that the guarantee does not apply to the update as a whole. In drives that do not guarantee sector atomicity, a power failure during an update can corrupt the sector being written.

Prefix semantics or *monotonicity* ensures that if an update survives failure and recovery, all its predecessors also survive. While no guarantees are made about how many writes will survive, prefix semantics can be combined with time limits or acknowledgments to let a host be sure that particular I/O operations have reached the media. Monotonicity can be quite useful in building reliable systems, but while disks routinely provide this property, file systems and disk drivers routinely remove it in the effort to improve performance.

Checkpoints allow the user to specify a consistent state to recover to. After recovery, the state is that of the latest complete checkpoint. Checkpoints are atomic, so that either all changes made before the last checkpoint are seen, or none of them are, and the state is rolled back to that of the previous checkpoint. Checkpoints allow the user to control the state of secondary storage so that the data always satisfies application-specific consistency criteria, simplifying recovery.

Transactions, typically provided by database systems, offer one of the strongest guarantees around. Transactions guarantee that entire groups of operations are atomic with respect to failure, and, furthermore, that there is some order in which the various transactions could have run, called a serial schedule, which is semantically identical with the actual system behavior. Transactions differ from checkpoints in that they can contain both reads and writes, and that there can be several transactions in progress simultaneously.

Mime provides support for all of these consistency guarantees. Straightforward use gives prefix semantics, and a multi-version visibility feature yields checkpoints and atomic snapshots, which can be combined with a concurrency control algorithm [Bernstein87, Barghouti91] to support full transactions.

1.2 Related work

Shadow-based recovery schemes have been used in databases for many years [Gray81, Reuter84, Bernstein87], but their use inside the disk subsystem to provide extended recovery semantics is new. Shadowing systems have recently coming into vogue in the file system community in the guise of log-structured file systems that never overwrite active data in place [Rosenblum92].

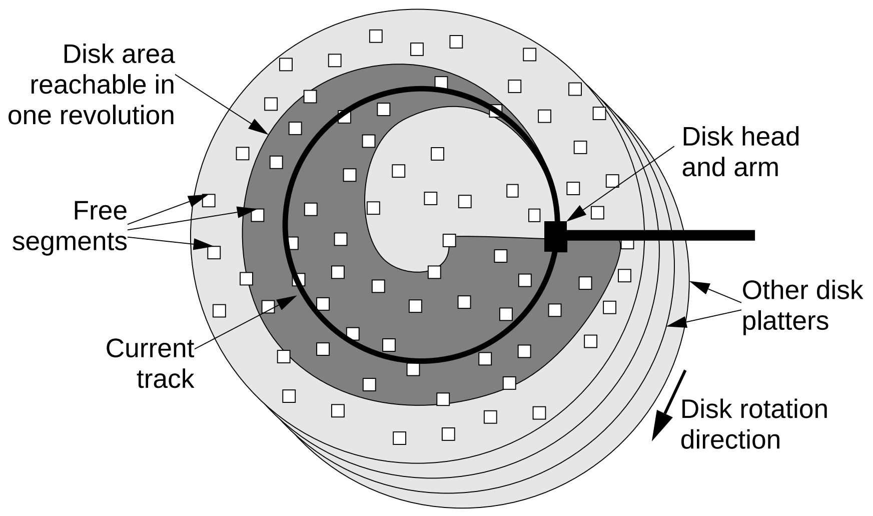

Figure 1: free blocks and the reachable area in a Loge disk device.

The techniques that we use to recover volatile memory images are extensions of those of [Birrell87] and [Lam91], although we add a new incremental checkpointing algorithm to their logging and recovery techniques. Our checkpointing technique is similar to ones proposed for large, memory-resident databases [DeWitt84, Eich87, Lehman87].

Mime is based heavily on our previous work developing Loge[3] [English92]. Since an understanding of that approach is essential to a feeling for how Mime works, we will describe it in some detail here.

A Loge disk reserves a small portion (typically 3–5%) of the disk as *free segments*, spread across the surface of the disk (Figure 1). When a write occurs, the disk selects a free segment near the current head position using a free map of the available segments and writes the data there. (There is almost always one close by: a typical 5.25” disk will have 15–19 data surfaces to select among, and with modern disks, it is possible to seek to roughly 40% of the disk’s sectors in a single revolution.) A record is kept in an indirection table of the location of the new block so that it can be found again on a read. The segment containing the old block is then freed, so that the number of free segments remains constant.

This approach substantially reduces write latency since there is typically a free block within a fraction of a disk revolution of the current head position. Simulation studies reported in [English92] using real disk traces showed write performances of 2.3, 2.4, and 2.6 times a regular disk, and overall performance increases of 25%, 20%, 36% respectively, once the slight degradation in read performance is taken into account.

In addition, because writes can be performed in any order without performance penalty, they can be performed in the order they are issued, maintaining monotonicity. Finally, since an update never

[3]. Loge (pronounced loh-ghee), is the Germanic god of fire.

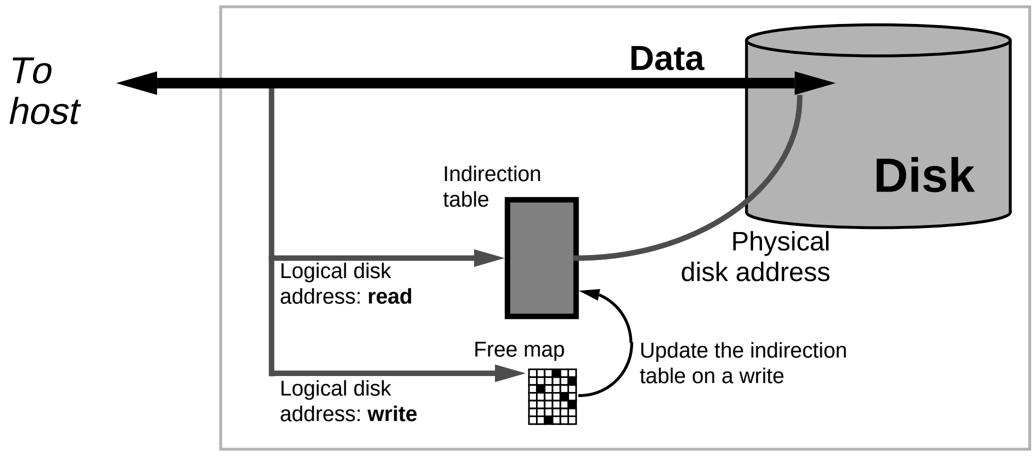
To host
Data
Disk
Indirection table
Logical disk address: **read**
Physical disk address
Free map
Update the indirection table on a write
Logical disk address: **write**

Figure 2: internal structure of a Loge disk device.

overwrites the old on-disk copy of the data, all updates are atomic. In Loge replaced blocks are put in the free pool, whereas in Mime these blocks serve as shadow copies containing the old data, and are exploited to provide the additional recovery facilities that are the subject of this paper.

Loge tracks data blocks with an indirection table which maps logical block numbers to their current physical locations (Figure 5). Additionally, a *free map* is used to find free blocks efficiently. To allow recovery should the in-memory indirection table be lost, Loge stores inverted indices as part of the data writes (i.e., at no extra cost) so that the table can be rebuilt by scanning the entire disk after a failure. This typically requires a little less than ten minutes; Mime improves on this recovery time by about a factor of ten.

The distorted mirrors of [Solworth91] use the indirect-write technique of Loge to provide fast writes and good sequential transfers, at the cost of a doubling of storage capacity. The performance numbers in that work were based on a simulation that failed to take the controller overheads and track switch / settle times into account: we avoid this (fairly significant) source of error, and also report on measured data from a real implementation of our ideas.

In a rather similar fashion to log-based file systems, Mime uses shadow copies to avoid having to do arbitrary seeks on a write. The read performance of the two is similar, except in the case of large, sequential transfers to data that was written close together.[4] The Mime architecture is optimized for an environment where frequent short writes are important: as is the case in many UNIX file systems [Ruemmler93], and some database applications. Unlike log-based file systems, Mime does not need non-volatile RAM to allow delayed writes to survive power failures and system crashes, and nor is it subject to the roughly 40% write-cost overhead of background cleaning [Rosenblum92].

4. Note that it is possible to augment Mime's free-space management policies to reserve multi-block runs of free space: these would be used to improve the performance of contiguous writes. Performance could asymptotically approach sequential disk transfer rates as the size and number of such segments increased, at the cost of lower effective disk capacity, just like LFS.

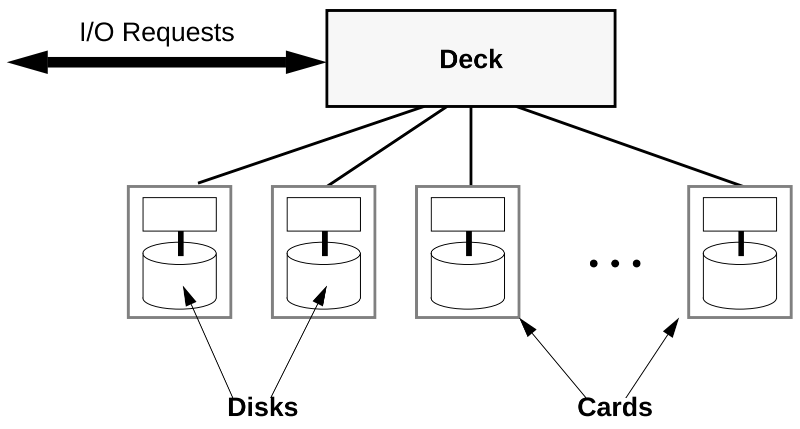

Figure 3: overview of the Mime functional architecture.

As with recent work on data shuffling to improve performance [Vongsathorn90, Ruemmler91], the data placement on a Mime disk can be adaptively modified in the background to approach the observed read access pattern. With a standard 4.2BSD file system layout, this approach can produce a 15% improvement in read performance; our numbers from simulating a Mime disk doing the same show slightly better results [Musser92].

2 Overview of Mime

Mime offers the same basic interface as a disk, extended with operations to support multiple views of data. This section describes the internal architecture of a Mime device, and introduces these additional operations.

2.1 The Mime internal architecture

A typical Mime implementation has a central controller node connected to between 4 and 16 special-purpose disk nodes (Figure 3). The controller node is called a *deck*; it has an I/O port to its host and contains roughly 1MB of RAM per 1GB of disk, or 32–128MB total (this perhaps 0.6% of the cost of the disk storage at current prices). The disk nodes are called *cards*: each contains 2GB of disk storage and a small amount of RAM (about 0.1MB). A typical card would be a regular disk drive with modified controller microcode that used part of the existing track buffers to support the Mime data structures.

Each card manages a single disk as an array of persistent, fixed-size storage *segments*. Each segment has an unique address and stores a *packet* consists of a data *block* and its metadata *tag* (Figure 3). Cards are designed so that tags are updated only if the associated data has been written in its entirety. This allows the tags to be used as reliable update flags by the deck. The card maintains a free map, similar to the one found in Loge, from which it allocates storage to packets. The free map is initialized by the deck.

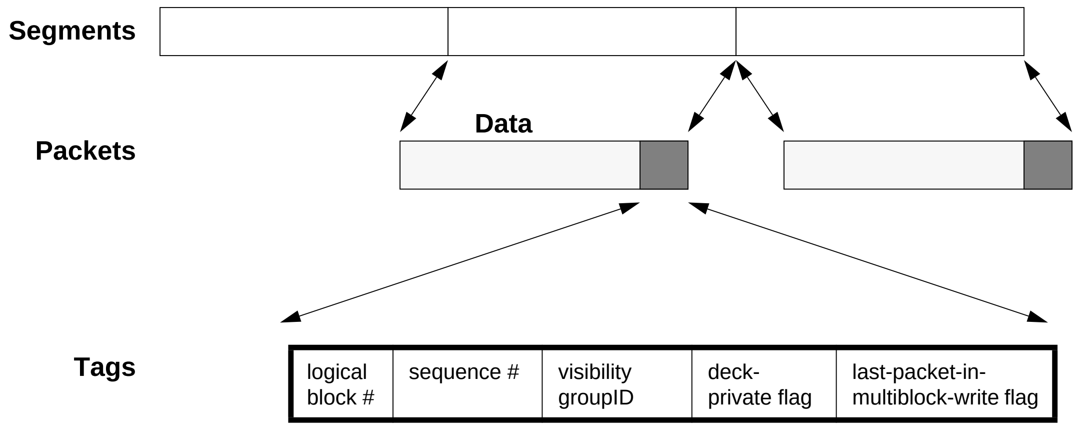

Figure 4: segments, packets, and tags.

The deck maintains a *primary index* that is similar to Loge's indirection table (Figure 5). This index maps logical block numbers to `<card#,segment#>` pairs. This index allows the deck to locate or relocate blocks between cards to improve performance. The deck is also responsible for consistency guarantees and recovery.

On a `read`, the deck looks up the card and segment numbers in its index, and forwards the request to the appropriate card. On a `write`, the deck selects the card that can complete the write the fastest, and dispatches the request to it. Upon completion, the card returns the new segment number, and the deck records the new location in its index. A `sync` operation causes any outstanding writes to be flushed to disk.

Deck data structures are periodically checkpointed. This is done incrementally, in a manner that allows the overheads in normal operation to be traded off against recovery time. On a failure, the most recent checkpoint of the deck and card data structures are recovered from the disks; the segments that have been written since the checkpoint are then scanned to determine which further operations have occurred; and the appropriate visibility and recovery operations are performed. Card data structures need not be checkpointed, since they can be reconstructed from deck data structures during recovery.

2.2 Mime operations

Mime supports three kinds of operations: *basic, visibility control,* and *synchronization control.*

2.2.1 Basic operations

Like any good storage device, Mime supports reads and writes: `read` takes a block number and returns the associated data; `write` takes a block number and a block of data, and records them for later retrieval. Mime also supports *multiblock* `reads` and `writes`. These take an initial block number and the number of blocks as their parameters.

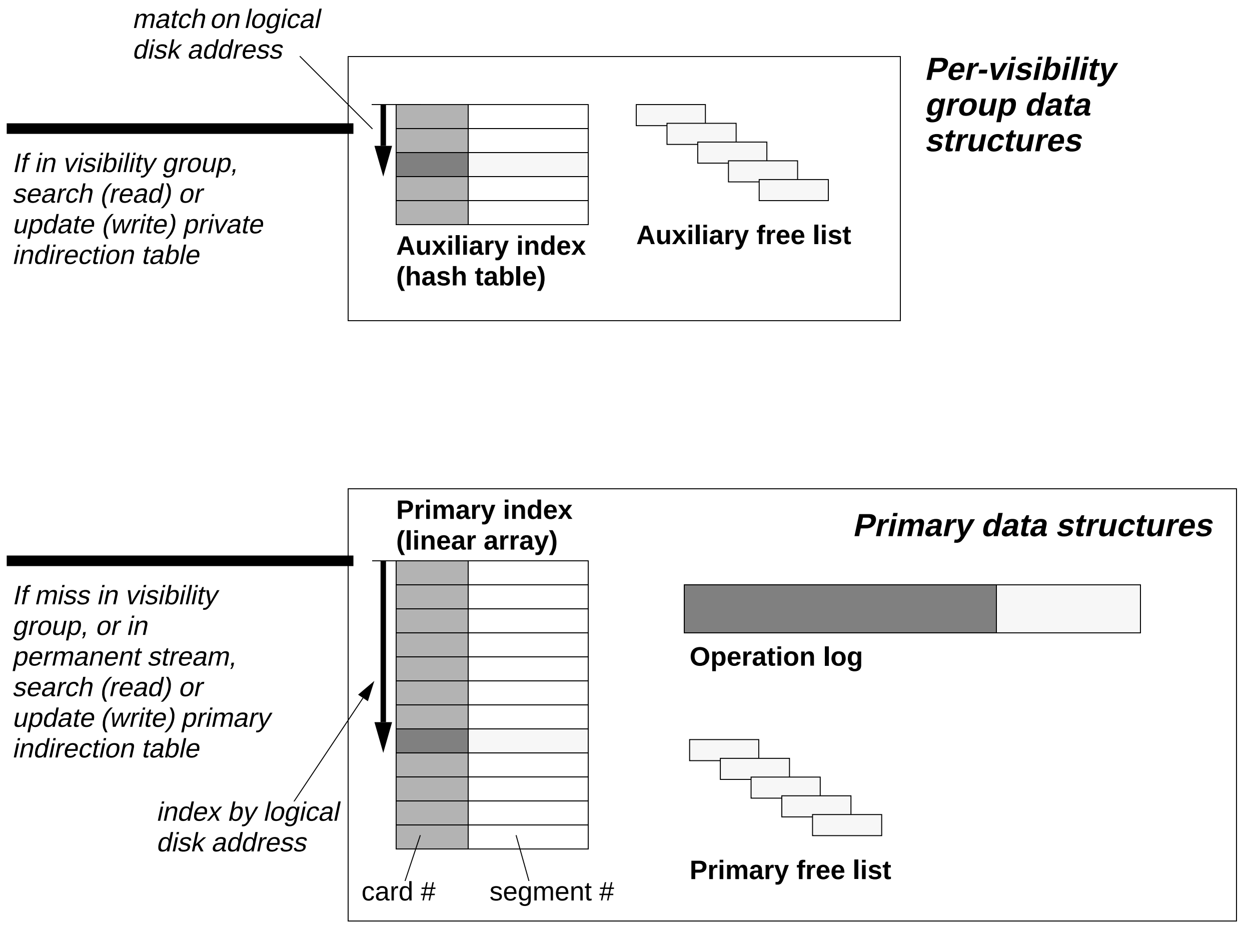
match on logical disk address
If in visibility group, search (read) or update (write) private indirection table
Auxiliary index (hash table)
Auxiliary free list
Per-visibility group data structures
Primary index (linear array)
Primary data structures
If miss in visibility group, or in permanent stream, search (read) or update (write) primary indirection table
Operation log
index by logical disk address
card #
segment #
Primary free list

Figure 5: summary of the Mime deck data structures

On Mime, all writes are atomic—even multi-block ones. In case of failure, Mime guarantees that either the entire write completed or none of it did. Since the new data is written into the free segment pool, the size of a multiblock write is limited by the number of free segments in the device.

2.2.2 Visibility control operations

The Mime write mechanism always leaves the old data on the disk after a write as a shadow copy. This can be used to provide a limited form of transaction support, as follows. A write operation can be labelled as belonging to an *visibility group*: such operations are called *provisional* because they do not appear to take effect outside the visibility group; operations that do not belong to a visibility group are called *permanent* because they are acted upon as received and cannot be undone. Writes within a visibility group are immediately visible within the group, but do not become externally visible until a `finish` is executed, at which time all provisional writes become visible atomically.[5] All writes in a visibility group can be rolled back with an `abort` operation. `New group`, `finish` and, `abort` are permanent operations: they can never be undone.

A host can get a new, unique visibility group number with the operation `new group`. There is also a `group status` operation that allows a host to find if a given visibility group is active, finished, aborted, or too old to have its completion status recorded.

Reads to a visibility group see the writes that were sent to it. They also see blocks written to the permanent command stream, unless a request is made to take a snapshot at the time the visibility group is created. If this is done, they will not see updates to blocks that occur outside the visibility group after it has been created. This technique is useful for obtaining consistent snapshots of the disk state—e.g., for on-line backup or to execute read-only queries in parallel with updates.

2.2.3 Synchronization control operations

Mime supports *operation streams* to allow interactions between commands to be controlled. There are separate operation streams for each visibility group plus one for the permanent operations. To allow fine control of consistency and permanence, two stream synchronization primitives are provided:

- A `barrier` operation establishes a checkpoint (barrier point) within a visibility group, but does not immediately force execution of the previous operations. After a crash, a visibility group will be restored to a barrier point: if there is no barrier point, the visibility group will be aborted.
- A `sync` operation in the permanent operation stream guarantees that all permanent operations issued before the `sync` will survive a crash. Within a visibility group, that guarantee extends only to those operations protected by a `barrier`.

Note that unlike a commit which implies both grouping and persistence, `sync` and `barrier` are distinct operations, with different meanings. `Sync` controls movement of data between volatile and persistent media, while `barrier` groups data into atomic units. This allows, for example, the host to specify that a set of operations must occur as a unit, without having to wait for synchronous disk operations. We believe that this is a new facility, not available elsewhere.

Mime guarantees monotonicity within the permanent operation stream. The permanent stream behaves as if every operation issued an implicit `barrier` after itself. Since a `barrier` is explicit within a provisional stream, monotonicity is not relevant except with regard to the `barrier` itself. One difference between Mime and other storage devices is that `sync` is the only synchronous update operation provided. In this Mime is more like a file system than a disk drive, but a file system that preserves monotonicity.

5. We use the term `finish` rather than commit to avoid the permanence normally associated with the latter: our synchronization model is such that permanence is not imported by the `finish` operation unless it is followed by a `sync`.

3 Applications

This section describes a few sample uses for a Mime disk subsystem. Doubtless we will develop more as we explore the Mime functionality space further.

In traditional 4.2BSD-based file systems [McKusick84], metadata writes (directories and inodes) are flushed synchronously to disk quite often in order to preserve the invariant that metadata is always at least as up to data as user data. This synchronous data flushing is expensive and common: on one local system, about 70% of the writes were synchronous; on another about 60% of the metadata writes were overwritten in less than 30 seconds. The traditional synchronization technique used by the 4.2BSD file system and its derivatives is to wait for the first request to complete before issuing the next. Mime can help avoid two problems here. Firstly, its fast, short, in-order writes mean that there is no advantage to trying to schedule write operations (e.g., with SCAN) and thereby risk reordering metadata writes after user data. In addition, the visibility groups can be used to implement an atomic directory, inode and data update: each such update is put into a separate visibility group, and a `finish` (not a `sync`) issued once the data has been generated to fill the newly-allocated (logical) disk space. This technique can avoid the synchronous updates altogether.

Another application is atomically appending to a log. Because Mime supports atomic multi-block writes, the log append can be of arbitrary size, and only a simple mechanism is needed to determine, on recovery, whether the entire append occurred or not. (The alternatives are to pre-erase the log area, or to include some form of checksum over the data, or a flag in every sector's worth of the update. A Mime client can simply use a sequence counter at a known offset early in the update.

Simple transactions (with undo capability) can be built very easily on top of Mime. The ready availability of such facilities will increase the robustness of systems: all too often, full database semantics are not required [Birrell87]. Given a simple lock manager to ensure the level of serializability desired, Mime provides essentially everything else through its visibility groups. It is even possible to perform internal checkpoints via `barrier` operations.

Consistent backups become almost trivial by using the snapshot option of a visibility group: the backup can proceed at whatever pace is convenient for it,[6] while all subsequent activity is hidden from its view. Once the backup completes, the visibility group is `aborted` (since it will have no writes in it), which will atomically free up any old copies of data that has since been rewritten.

For log-based file systems, like Episode, the use of fast short writes makes it possible to consider extending the recovery semantics to user data, as well as file system metadata. (The barriers Mime provides allow tight ordering constraints to be enforced between metadata log updates and the data that they refer to. The regular Episode design sacrifices this in the interests of avoiding too many synchronous operations.)

6. Modulo the free space filling up.

Although we cannot easily provide the same raw data rate as LFS achieves with its 1MB segment writes, Mime does provide much greater crash-resilience because it does not need to retain data in volatile memory for long enough to build up a segment. (The cost of the Mime volatile RAM memory is roughly a sixth of that required for the non-volatile RAM needed for similar crash resilience in LFS [Baker92].) Also, [Baker92] goes on to observe that without such non-volatile RAM, many LFS writes are of short segments anyway, because of the effect of user-level `fsync` calls. And finally, the long writes that give LFS its performance edge can increase read latencies [Carson92].

In OLTP applications such as the TPC-A and TPC-B benchmarks, which are dominated by random I/O traffic, roughly half of all the I/Os are writes caused by replacing a dirty page in the database buffer cache. The Mime write algorithm will reduce the total I/O time—and thus increase the transaction rate—by about 25%, given that writes go roughly twice as fast as with a regular disk, and reads take the same amount of time. (In this application, the layout changes introduced by shadowing the data pages are not of concern, since the pages are always randomly accessed anyway.)

4 Implementation

Now that the basic approach has been outlined, the details can be fleshed out. Since the recovery procedure affects normal operation, we give a quick overview of it first. The discussion then moves to operations within the permanent operation stream, followed later by the additional facilities needed to support provisional operations.

4.1 Recovery from failures

The primary index is stored in volatile storage and changes very rapidly, so that it is necessary to reconstruct the index after a failure. This recovery takes place in two stages: first, the system recovers to a checkpoint and then it replays the operations following the checkpoint, using inverted indices within stored packets to determine what operations took place. The details of the checkpoint procedure are not important at this point, but it is important to note that searching the disk (as in the case of Loge) can be made much more efficient if only the segments that could have been written since the last checkpoint need to be searched—and this set is precisely the list of free blocks at the time of the checkpoint. For this reason, Mime delays the return of segments to the free map until checkpoint time. In addition, since Mime supports operations (such as `finish` and `barrier`) that do not result in packet writes, these operations must be recorded through a separate mechanism. For this, Mime maintains an *operations log*, containing all synchronization and visibility operations since the last checkpoint. This log is written to disk at every `sync`, into a packet with a special metadata tag, and is recovered during the same scan of free blocks that recovers the data writes. (Notice that this is quite cheap, since syncs can be made rare: with Mime, they are only needed to perform a "make this persistent against crashes" operation, not the ordering provided by barriers. They correspond roughly to points at which a group commit is taken in a database system.)

4.2 Deck data structures and operations

The primary index is a simple linear array requiring about four bytes for each block of data stored. A 16GB Mime device (i.e. about 8 disks) requires about 16MB of deck RAM for its index. With RAM costs approaching $30 / MB, and disks $5 / MB, the deck memory represents about 0.6% of the system cost.

When a disk segment is overwritten, the old segment is placed in a *free list*, and held there until the next data structure checkpoint. Freeing segments prior to the checkpoint might result in cards writing to segments not listed in the checkpointed index, which could cause updates to be lost during recovery. As part of the checkpoint, the free list is transferred to the card and incorporated into the free map, making the segments available for reuse.

The deck issues sequence numbers for all update operations. The permanent stream sequence numbers are contiguous so that gaps can be detected during recovery. Each visibility group has its own sequence number set, independent of the permanent stream. Since `new group`, `finish`, and `abort` are part of the permanent command stream, they receive permanent stream sequence numbers, as well. Multiblock writes are allocated contiguous sequence numbers, one for each block. The highest-numbered block in any write is flagged so that partial multiblock writes can be detected cheaply and discarded during recovery.[7] `New group`, `barrier`, `finish`, and `abort` operations are recorded in the operation log.

A `sync` causes the operation log and any outstanding writes to be sent to disk immediately.

4.3 Card data structures and operations

The card is responsible for storing packets on disk. As described above, the key property that must be maintained is that it must only modify a packet's tag value after the packet's data is successfully written. This can be accomplished by extending disk sectors with enough additional bytes to store tags with data in the same sectors. Many modern disks allow the sector size to be increased slightly beyond a power of two for just this reason. Another alternative would be to store tag information in non-volatile RAM, a slightly more costly option, but one that would allow faster recovery. Other alternatives are possible, but are generally more complicated or less efficient.

The card's primary data structure is its free map: a bitmap indicating which blocks are available for writing to. For a 2GB disk with 4KB segments, the free map will be about 64KB in size. The operations provided by a card are quite straightforward, and described in Table 1.

4.4 Visibility groups

For each active visibility group, the deck creates two data structures: (1) an auxiliary index (a small hash table) to locate provisionally written blocks, and (2) an auxiliary free list. `Read` within a visibility group look first in the auxiliary index, and then in the primary one. `Write` updates only the visibility group's index and free list. `Barrier` is logged in the global operation log and the fact that a `barrier` occurred at a particular time is recorded in the auxiliary table.

[7]. The "last block" flag is equivalent to a `barrier` operation completing the multi-block write.

Table 1. Operations on a card.

operation	*effect*
read (segment#) → packet	retrieve data from a segment
write (packet) → segment#	writes a <data,tag> pair in a free segment (as marked in the free map) and returns its address. The segment is removed from the free map
free (segment#)	marks the free map entry for a segment so that it can be written. A card never marks a segment free—only the deck
update (segment#, packet)	stores a <data,tag> pair into the given segment (only used for writing out deck data structure checkpoints)
read_tag (set of segments) → set of tags	read the tags associated with a set of segments
download_freemap (freemap)	causes the card to replace its freemap with the provided one.

On a `finish`, a record is added into the operation log (it doesn't have to be written immediately to disk unless it is followed by a `sync`). Then, the deck merges the visibility group's auxiliary index and free list with its primary ones, and discards the auxiliary data structures. On an `abort`, a record is added into the operation log. All segments in the visibility group's hash table are added to the free list, and the auxiliary data structures discarded.

If a visibility group requests that a snapshot be created, write operations to the permanent command stream cause pointers to old segments to be inserted into the visibility group's auxiliary tables, so that reads relative to the visibility group do not see the changes. Blocks inserted into the table in this manner are flagged so that they will be freed when the visibility group terminates, instead of merged into the primary index.

Mime's externally-visible operations are summarized in Table 2.

4.5 Incremental checkpointing

The data structures described in the previous section reside in volatile memory and are destroyed by power failures. In order to recover from such failures, Mime creates incremental data structure checkpoints. These are written to static, fixed areas of the disks so that they can be easily located during recovery.

The memory area containing the data structures to be protected is divided into *k* sections. Each checkpoint record (*metablock*) contains two parts—a snapshot of one of the *k* sections and a log of recent changes to the entire memory area. So that our logging facility is independent of the data structures stored, Mime logs memory update operations, rather than recording changes to logical structures. Every change is recorded in the log as a <memory address, new value> pair. Whenever the log area becomes full, a metablock is constructed by advancing a pointer to one of the *k* memory sections, constructing a metablock out of the log and the new memory section, and writing it to the disk.

Table 2. Mime deck operation summary

Operation	*Permanent*	*Provisional*
Read	Search main index for card/segment# Read segment	Search auxiliary index If miss, search main index Read segment
Write	Write segment to card Update main index with returned segment# Add old segment to free list *or* insert segment into auxiliary indices keeping snapshots.	Write segment to card Update auxiliary index with returned segment# Add old segment to auxiliary free list
Finish	Put finish record in operation log Move writes from auxiliary index into main index Merge auxiliary free list into primary one Discard auxiliary data structures	N/A
Abort	Put abort record in operation log Put all write segments in auxiliary index into auxiliary free list Merge auxiliary free list into primary one Discard auxiliary data structures	N/A
Barrier	N/A	Put barrier record into operation log
Sync	Finish all outstanding permanent writes and finished provisional writes Write operation log to card; add log segment to free list	Finish all outstanding writes in this group (up to the last barrier) Write operation log to card; add log segment to free list

At any given time, the *k* most recent metablocks can be used to recover the entire structure as of the last checkpoint. The oldest metablock is read first and its contents copied into main storage. As each subsequent metablock is read, its log is replayed against the entire memory area. After this is done *k* times, each memory section will have had an image restored, as well as a log replay for all changes following the time the image was saved. The fact that a checkpoint can be interrupted in the middle of a write—leaving neither the old nor the new data valid—requires that the location being overwritten not contain valid data, which implies that at least $k + 1$ metablock storage locations on the disks.

This technique allows control over the size of the log, and thus the amount of time needed for recovery. Increasing the fraction of each metablock used for logging decreases the frequency of checkpoints and the total logging bandwidth, but increases the size of the log, and as a result, the recovery time. Conversely, increasing the portion of the metablock devoted to checkpoints increases the load the checkpointing places on the system, but reduces the recovery time. Suppose, for example, that a metablock were 32KB in length, and that the metablock was split between log and memory image. If each 4KB write required 16 bytes of log information, then the metablock would have to be flushed after every 1000 write operations, for an overall system load of less than one per cent. With 2GB cards, each card would need to recover about 2MB of data, which would be 250 full-track reads, or about 4 seconds. Since cards can operate in parallel, this stage of recovery will probably be limited by the deck, rather than the cards.

4.6 Recovery

The recovery-related fields of these tags are shown in Table 3.

Table 3. Fields in a tag.

field in tag	*size*	*purpose*
block number	7 bytes	names the block a segment's data belongs to
sequence number	8 bytes	monotonic counter used to determine the most recent version of a block with multiple copies
visibility group number	8 bytes	visibility group that this block was written in
deck data structure flag	1 bit	identifies segments used by the deck for data structure checkpoint operations
last block flag	1 bit	flags highest-sequenced block in a (multiblock) write

The incremental checkpointing operations are used to save the state of Mime memory containing primary data structures. The recovery algorithm recovers past this point by using a combination of information stored in the checkpoints and the tags of the blocks that have been written since the last checkpoint. The free list ensures that no such block will be deleted and reused until after the next checkpoint completes. Here is the complete recovery algorithm:

1. *Restore the state of the deck data structures to the last checkpoint.*
   Determine which are the *k* most recent, valid metablocks, and then restore the state of the internal deck data structures to that of the last incremental checkpoint. This will involve reading about 8MB from each card, and will take about 4 seconds at a 2MB/s net transfer rate since it can be performed in parallel across all the cards.
2. *Recover all tags written since the last checkpoint.*
   Retrieve the tags of all the free segments on each card. On a 2GB card with 3–5% free space, this step will take 50–100 seconds (rather than the15 minutes or so of Loge).
3. *Read the operations log for operations performed since the last checkpoint.*
   Identify and retrieve all valid operation logs. All valid write operations are recovered from tags written since the last checkpoint; `finish`, `abort`, and `barrier` are recovered from the operation log. All operations are sorted into different lists according to the command stream to which they belong, in ascending sequence number order.
4. *Discard all incomplete multiblock writes.*
   These are indicated by there not being a `last block` flag, or by a sequence number within the write being missing.
5. *Recover the permanent command stream.*
   This has to occur before the visibility groups are recovered because `finish` and `abort` are permanent operations. Redo the permanent operations in the operations log until a gap in a sequence number is found. This ensures monotonicity. All normal data structure operations that would occur in regular processing occur here, but no changes are made on disk: writes are added into the primary index; finishes and `abort`s of

visibility groups are performed as they are encountered.[8] Entries are added to auxiliary tables with snapshots to record addresses for overwritten blocks.

6. *Recover active visibility group data structures.*
   Locate the last `barrier` within a visibility group. Add all writes prior to the `barrier` to the auxiliary index, and discard any subsequent writes to the group. If the visibility group has no `barriers`, abort the group.
7. Initialize free maps. Now recovery is complete and we can resume normal operations.

Most of this process involves only main-memory operations, and executes quickly: step 2 is the performance limiter.

5 Conclusions

We present our concluding remarks in three parts: a review of the functionality the Mime architecture provides; a description of the state of our prototype implementation we built to validate our architecture; and a summary of the lessons to be learned from this endeavor.

5.1 Analysis

Any implementation that provides failure guarantees does so on the basis of assumptions about the failure modes of its components. In this discussion we assume no irrecoverable media failures and fail-stop behavior for all active system components. With these assumptions, Mime provides support for all the previously-described recovery semantics:

- *Write enqueueing* is handled as soon as the operations are issued to Mime.
- All writes are *atomic*, which simplifies a number of operations. Client file systems, for example, can be written without concern for partial writes.
- *Monotonicity* is provided by implicit `barriers` in the permanent operation stream: recovery will be to the latest `write` operation completed.
- *Checkpoint semantics* are provided by `barriers`. Since `syncs` are separate operations, a Mime client never has to wait for an operation to complete merely to achieve reliable ordering behavior.
- *Transactions* are supported by the visibility groups and the `finish` and `abort` operations. While full transaction semantics cannot be provided at this level—they require locking and synchronization at higher levels in the system to provide serializability—Mime provides the necessary atomicity and undo capabilities.

It is worth noting that these additional capabilities are essentially free in Mime, due to the high-performance shadowing system that supports them. In Mime, the shadowing *improves* the performance of normal disks. The additional capabilities built on top of shadowing merely supply better functionality on top of the increased performance.

[8] The next step describes how to accomplish this: the visibility group has to be (partially) recovered before it can be aborted or finished. If a `finish` cannot be completed because not all the visibility group writes have completed, the `finish` record is declared “missing”: the visibility group is rolled back to its last `barrier`, and the permanent stream has reached its prefix recovery point, so no further recovery processing is required.

5.2 A prototype implementation

We have validated the practicality of the Mime architecture by constructing a single-disk prototype. This prototype is fully functional except for the snapshot facility. It is built from storage nodes with an Inmos T800 transputer processor, 4MB of memory and an local SCSI-attached HP97560 1.3GB 5.25” disk. The nodes are also fitted with hardware that allows them to acquire rotational position information to within a few microseconds by using the disks’s built-in spin-synchronization mechanism. An additional node is dedicated to communication to a host, and programmed to appear as if it is a regular SCSI disk drive with the Mime extensions.

As predicted by our modelling, we achieved significant performance improvements. For example, we ran some experiments to perform 4KB writes, across a 10-cylinder range. (This restriction on the seek distance models the locality seen within a cylinder group; it only serves to make the non-Mime case perform better.) The overhead for the prototype to act as a non-Mime disk controller was 0.23 ms per request, with a mean physical disk time (including SCSI bus transfers) of 15.05 ms.

The Mime algorithms increased the controller overhead by 0.63 ms (perhaps 600–1000 instructions on our roughly 2MIPS processors; about the same as a regular disk controller’s overheads), but resulted in the physical disk time dropping to only 7.92 ms. The net gain was thus 6.5ms—a reduction in total write time of 44%.

The shadow paging we are using in Mime for recovery is actually *improving* the disk system performance compared to a regular disk. Since almost all of the extra Mime operations are memory-based rather than disk-based, we see a net gain in performance despite the additional functionality we provide.

5.3 Summary

Mime uses a single, consistent shadowing scheme that comes for free as a result of the write-latency optimizations introduced by Loge. This allowed us to streamline our algorithms considerably over those used in conventional redo / undo systems. The use of incremental data structure checkpointing allows us to provide an efficient recovery mechanism with low overhead in normal operation. By doing this at low levels in the storage subsystem, we are able to achieve improved performance at the same time as strong recovery guarantees.

Our Loge simulation results from [English92] show that a controller embedded in the disk (as opposed to at the other end of a SCSI bus) should be able to achieve 3–5ms for physical disk times, plus an unquantified amount of controller overhead (probably less than 1ms). On a detailed trace-based simulation of a month’s worth of disk accesses to a 4.2BSD-based file system, we saw write performance increase by more than a factor of two.

The robustness properties that are the main thrust of the Mime architecture come almost for free given the underlying shadow writing mechanism: they add a small amount of controller overheads, and an occasional checkpoint or operation log write, but have the side effect of eliminating the need for most synchronous writes to the disk.

Mime has been presented here as an architectural approach to the design of dedicated disk subsystems. With less immediate feedback from the disk to the host about its rotation position (as in our prototype), we have shown that it is also possible to use these techniques in a host-based disk driver. The result is somewhat smaller gains in performance, but identical functionality.

Acknowledgments

Mime is being developed in the context of the DataMesh research project [Wilkes89, Wilkes91], which is designing parallel system storage architectures for high performance, availability, and scalability.

Bart Sears and Chris Ruemmler provided valuable feedback on drafts of this paper.

References

[Baker92] Mary Baker, Satoshi Asami, Etienne Deprit, John Ousterhout and Margo Seltzer. Non-volatile memory for fast, reliable file systems. *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems* (Boston, MA). Published as *Computer Architecture News* **20** (special issue):10–22, 12–15 October 1992.

[Barghouti91] Naser S. Barghouti and Gail E. Kaiser. Concurrency control in advanced database applications. *Computing Surveys* **23**(3):269–318, September 1991.

[Bernstein87] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems*. Addison-Wesley, Reading, Massachusetts, 1987.

[Birrell87] Andrew D. Birrell, Michael B. Jones, and Edward P. Wobber. A simple and efficient implementation for small databases. *Proceedings of 11th ACM Symposium on Operating Systems Principles* (Austin, Texas). Published as *Operating Systems Review* **21**(5):149–54, November 1987.

[Carson92] Scott Carson and Sanjeev Setia. Optimal write batch size in log-structured file systems. *USENIX Workshop on File Systems* pages 79–91 (Ann Arbor, MI), 21–22 May 1992.

[Case92] Brian Case. DEC's Alpha architecture premiers: 64-bit RISC architecture streamlined for speed. *Microprocessor Report* **6**(3):10–14. Micro Design Resources Incorporated, 4 March 1992.

[Dasgupta91] Partha Dasgupta, Richard J. LeBlanc, Jr., Mustaque Ahamad, and Umakishore Ramachandran. The Clouds distributed operating system. *IEEE Computer* **24**(11):34–44, November 1991.

[Dewitt84] David J. DeWitt (et al). Implementation techniques for main memory database systems. *Proceedings of 1984 SIGMOD Conference on Management of Data* (Boston, MA), June 1984.

[Eich87] Margaret H. Eich. A classification and comparison of main memory database recovery techniques. *Proceedings of International Conference on Data Engineering*, pages 332–9. IEEE, 1987. Reprinted in *Parallel Architectures for Database Systems*, A. R. Hurson, L. L. Miller and S. H. Pakzad, editors (IEEE Computer Society Press, 1988).

[English92] Robert M. English and Alexander A. Stepanov. Loge: a self-organizing storage device. *Usenix Technical Conference* (San Francisco, Winter'92), pages 237–51. Usenix, 20–24 January 1992.

[Gray81] Jim Gray, Paul McJones, Mike Blasgen, Bruce Lindsay, Raymond Lorie, Tom Price, Franco Potzulo, and Irving Traiger. The recovery manager of the System R database manager. *Computing Surveys* **13**(2):223–42, June 1981.

[Kondoff88] Alan Kondoff. The MPE XL data management system exploiting the HP Precision Architecture for HP's next generation commercial computer system. *Digest of papers, Spring COMPCON'88* (San Francisco, CA), pages 152–5. IEEE, 29 February–4 March 1988.

[Lam91] Kwok yan Lam. An implementation for small databases with high availability. *Operating Systems Review* **25**(4):77–87, October 1991.

[Lee89] Ruby B. Lee. Precision Architecture. *IEEE Computer* **22**(1):78–91, January 1989.

[Leffler89] Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, and John S. Quarterman. *Design and implementation of the 4.3BSD UNIX operating system*. Addison-Wesley, 1989.

[Lehman87] Tobin J. Lehman and Michael J. Carey. A recovery algorithm for a high-performance memory-resident database system. *Proceedings of Association for Computing Machinery SIGMOD 1987 Annual Conference* (San Francisco, May 27–29, 1987). Published as *SIGMOD Record*, pages 104–17, May 1987.

[Reuter84] Andreas Reuter. Performance analysis of recovery techniques. *ACM Transactions on Database Systems* **9**(4):526–59, December 1984.

[Rosenblum91] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *Proceedings of 13th ACM Symposium on Operating Systems Principles* (Asilomar, Pacific Grove, CA), pages 1–15. Association for Computing Machinery SIGOPS, 13 October 1991.

[Ruemmler91] Chris Ruemmler and John Wilkes. Disk shuffling. Technical report HPL–91–156. Hewlett-Packard Laboratories, Palo Alto, CA, Oct. 1991.

[Ruemmler93] Chris Ruemmler and John Wilkes. UNIX disk access patterns. *Proceedings of the Winter 1993 USENIX* (San Diego, CA, 25–29 Jan. 1993), pages 405–20, Jan. 1993.

[Scott90] Michael L. Scott, Thomas J. LeBlanc, Brian D. Marsh, Timothy G. Becker, Cezary Dubnicki, Evangelos P. Markatos, and Neil G. Smithline. Implementation issues for the Psyche multiprocessor operating system. *Computing Systems* **3**(1):101–37, Winter 1990.

[Seltzer90] Margo Seltzer and Michael Stonebraker. Transaction support in read optimized and write optimized file systems. *Proceedings of 16th International Conference on Very Large Data Bases* (Brisbane, Australia), 13–16 August 1990.

[Solworth91] Jon A. Solworth and Cyril U. Orji. Distorted mirrors. In *Proceedings of 1st International Conference on Parallel and Distributed Information Systems* (Miami Beach, FL), pages 10–17, 4–6 December 1991.

[Vongsathorn90] Paul Vongsathorn and Scott D. Carson. A system for adaptive disk rearrangement. *Software–Practice and Experience* **20**(3):225–242, March 1990.

[Wilkes89] John Wilkes. DataMesh—scope and objectives: a commentary. Technical report HPL–DSD–89–44. Distributed Systems Department, Hewlett-Packard Laboratories, 18 July 1989.

[Wilkes91] John Wilkes. DataMesh—parallel storage system for the 1990s. *Digest of papers, 11th IEEE Symposium on Mass Storage Systems* (Monterey, CA), pages 131–6, 7–10 October 1991.