

# ***Lumberjack: A Log-Structured Persistent Object Store***

David Hulse, Alan Dearle, and Alastair Howells

Department of Computing Science and Mathematics  
University of Stirling, Stirling, FK9 4LA, Scotland

{dave, al, aqh}@cs.stir.ac.uk

## **Abstract**

*Lumberjack is a log-structured persistent object store intended for use with conventional operating systems such as Unix. The design draws together facets of previous work, in particular, the CPOMS object store used to support PS-algol, and the page-based log-structured store used in the Grasshopper persistent operating system. Lumberjack exhibits a novel store design that provides a number of benefits: i) an efficient commit strategy with respect to meta-data, ii) the avoidance of installation reads and writes during commit, iii) support for incremental parallel garbage collection. Additionally, log-cleaning and garbage collection may be integrated providing the ability to re-cluster data to exploit locality. We see Lumberjack as delivering object-based logging for use in systems such as Napier88 and persistent Java, thereby providing an alternative to the traditional shadow-paged stores used in these systems.*

## **1. Introduction**

*Lumberjack* is a new log-structured persistent object store intended for use with conventional operating systems such as Unix. The design of *Lumberjack* draws together facets of previous work, in particular, the CPOMS object store [3] used to support PS-algol [1], and the page-based log-structured store [7] used in the Grasshopper persistent operating system [13].

We believe that *Lumberjack* exhibits a novel store design that provides a number of benefits. Firstly, the use of *log-structured* storage techniques allows I/O operations to be optimised by performing them sequentially rather than as a series of random accesses. In doing so, the cost of seeking and rotational latency is virtually eliminated which results in a significant increase in performance when compared to other forms of storage such as a standard file system [14]. The design of *Lumberjack* exploits this increase in performance by providing an extremely efficient commit strategy with respect to the meta-data required in the management of the store. As described in section 4.3, very little meta-data needs to be written to stable storage during a *snapshot* operation. The result is that almost 100% of the time taken to perform a snapshot can be attributed to writing modified object data to the log. In contrast, the cost of snapshot operations in stores based on shadow paging can easily be dominated by the overhead of writing back the meta-data [6].

Many systems [5, 8, 15], technically known as *log-based* systems, store objects in a fixed location on disk and use a log to increase the efficiency of snapshot operations by writing data sequentially rather than randomly as would otherwise be required. The data written to the log is copied to the appropriate disk locations asynchronously which requires additional read and write operations to be performed. In contrast, the *Lumberjack* store adopts a *log-structured* approach in which objects do not have a fixed location on disk. Instead, the store is applicative in nature in that no stable data is overwritten and the current version of an object is defined to be the most recently logged version. Once an object is written to the log, it is stable and potentially recoverable. The key difference between the two approaches is that a log-structured system unifies the object database with the log thereby avoiding the need for asynchronous updates.

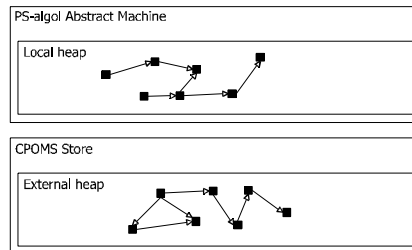
A further advantage of the use of log-structured storage techniques stems from the manner in which free space is managed within the log. As new versions of objects are written to the log, the previous versions of those objects are rendered obsolete. Thus, the log requires periodic garbage collection, also known as *cleaning*, to reclaim space. In the case of *Lumberjack*, this can occur in parallel with the normal operation of the store. Furthermore, since cleaning involves the relocation of *live* objects, the cleaning process provides an ideal opportunity to re-cluster objects to take advantage of physical locality. In addition, it is possible to exploit the actions of the cleaner to facilitate application-level garbage collection as is required in many systems such as *Napier88* and persistent Java (see section 4.3.7).

A final contribution of the store is that it permits flexibility in the policy used to perform snapshot and recovery operations. In particular, both eager and lazy write-back strategies may be employed to control the writing of store meta data. That is, the write back of meta-data may be delayed until quiescent periods thus permitting the opportunistic use of unused I/O bandwidth to improve performance.

The remainder of this paper is structured as follows. Section 2 describes the object addressing technique used within the CPOMS architecture and section 3 describes the architecture of the page-based log-structured store used previously in Grasshopper. Section 4 presents the hybrid architecture that constitutes *Lumberjack*. Finally, section 5 describes the future directions for *Lumberjack* before concluding.

## 2. Object Addressing in the CPOMS

The CPOMS [3] is a persistent object management system written in C that provides a stable storage layer for the PS-algol abstract machine [1]. The architecture of the abstract machine and the CPOMS is shown in Figure 1, which depicts two heaps: a *local heap* that is managed by the abstract machine, and an *external heap* that is managed by the CPOMS. Objects held within the *external heap* are stable and are addressed by special pointers known as *persistent identifiers* (or PIDs). Such objects cannot be directly addressed by the abstract machine and must be copied into the *local heap* before use. The abstract machine addresses objects within the *local heap* using *local object numbers* (or LONs) which are equivalent to standard pointers.



**Figure 1:** The heap architecture of the PS-algol abstract machine and the CPOMS.

Objects copied into the *local heap* may contain PIDs that refer to other objects. When the abstract machine encounters a PID, the referenced object is copied into the *local heap* and the referring PID is replaced by the LON of the copied object. A table known as the PIDLAM is used to track the objects copied into the *local heap* to ensure referential integrity. At commit time, the LONs in each object are replaced by the original PIDs before the objects are copied back to the *external heap* where they are made stable. This technique of temporarily replacing PIDs with LONs is known as *pointer swizzling* [10].

### 2.1 External Heap Architecture

Within the *external heap*, objects are grouped within larger objects called *partitions*, each of which has a unique number. An object's PID reflects this grouping by way of its internal structure which comprises a *partition number* and an *object number* field. The object number is a 16-bit value and must be interpreted relative to the partition number, which is a 15-bit value<sup>†</sup>. Thus, the external heap can support up to 32768 separate partitions, each of which can hold up to 65536 objects.

Partitions are stored within *databases*, which are implemented as a pair of files. The first file, the *data* file, holds all of the objects within the partitions allocated to the database. The second file is an *index* and contains the file offsets of the objects within the data file. When a partition is allocated to a database, it is assigned a contiguous section of the index file through which the objects within the partition can be located. Each database initially contains a single partition and new partitions are added only when all previously allocated partitions are full of objects. Thus, new partitions are always allocated a region at the end of the index file and the offset of this region is always a multiple of 65536 entries.

The CPOMS must be able to locate an object using its PID, a requirement that is supported by a data structure known as the *Partition TO Database and Index map* (PTODI). The PTODI is an array of structures, each of which contains three fields: a partition number, a database number, and the offset of the partition's index region within the database. The array is ordered on the partition number field so that a binary search may be used to find the entry for a particular partition. The procedure used to locate an object given its PID is shown in Figure 2.

In step 1, the partition number is extracted from the object's PID and used to locate the partition's entry in the PTODI. Step 2 uses the database number from the PTODI entry to locate the correct database files. In

<sup>†</sup> The remaining bit in the PID is always set to 1 to enable the interpreter to distinguish PIDs from LONs.

practice there is another data structure used to map database numbers onto filenames although this is not relevant to the present discussion and has been omitted from the diagram for simplicity. Once the correct database has been identified, the object number from the PID is added to the index offset held in the PTODI to yield the *database object number* (steps 3 and 4). In step 5, this number is used to read the index file entry, which contains the offset of the object within the data file. The contents of the object may then be loaded from the data file into the local heap maintained by the abstract machine (step 6).

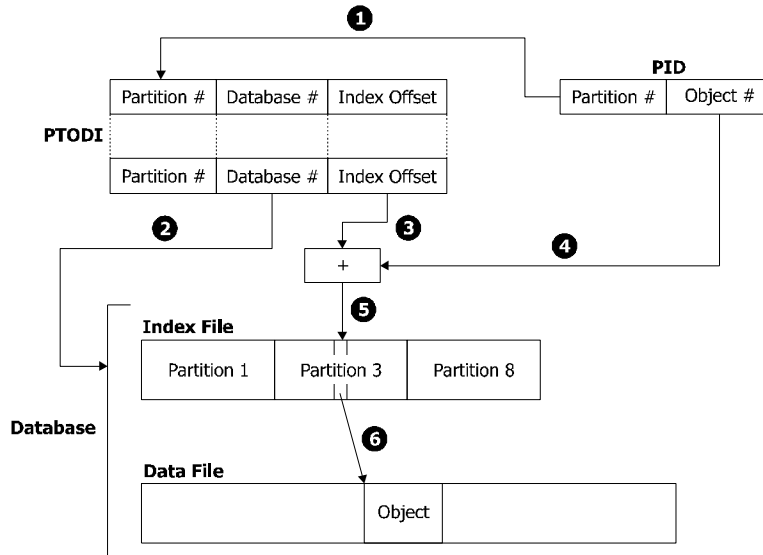


Figure 2: Object addressing within the external heap.

### 3. Log-Structured Storage in Grasshopper

In Grasshopper [13], all instances of the abstractions supported by the kernel are persistent. Internally, the kernel maintains a persistent virtual address space in which the meta-data representing each of the extant abstractions is held. This address space is paged to and from a log-structured persistent store managed by a *log server* [7]. The log server is also used to support the storage of user-level persistent data held within instances of the kernel abstractions. Thus, the log server provides Grasshopper with a centralised logging service that is used by a number of clients.

The interface to the log server supports the concept of a *logical log*, an abstraction that encapsulates the semantics of log-structured storage in a secure manner. In practice, each client of the log server typically uses one or more logical logs to store persistent data. Within a logical log, data is held in *segments*, which are the basic units of I/O and storage allocation. The use of segments was inspired by the *Sprite log-structured file system* [14] in which they are primarily used to ease the management of free space in the log. This is achieved by clustering log records of varying size within segment buffers held in main memory. As buffers become full, they are appended to the tail of the logical log using a single sequential write operation. When a segment no longer contains useful data, its space is reclaimed for use in subsequent write operations.

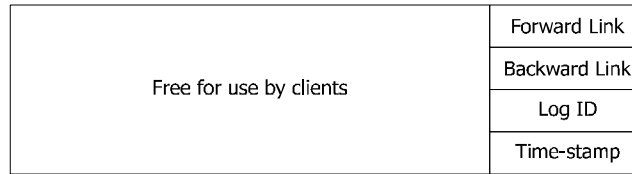
After some time, the locations of free segments will be scattered across the storage medium. Thus writing a new segment will require a seek operation followed by a sequential write. Since logging is supposed to exploit the improved bandwidth arising from the use of sequential I/O, the fact that seek operations are required to write segments seems to be at odds with this. However, by ensuring that size of segments is sufficiently large, the overhead of seeking becomes negligible. The implementations of the log server in both Grasshopper and *Lumberjack* use a segment size of 512 KB, which is considered to provide a reasonable trade-off.

The following sections describe the design and implementation of the log server. As noted in section 4, this design is used largely unchanged as the basis of the *Lumberjack* store. The version of the log server used by *Lumberjack* provides a slightly cut-down interface compared to that used in Grasshopper. For brevity, the interface used by Grasshopper has been omitted from the description that follows.

#### 3.1 The Internal Structure of the Log Server

Internally, the log server maintains a single *physical log* that is represented as a sequence of *segments*. From the point of view of the log server, a segment is simply a contiguous region of secondary storage that is identified by

the address at which it is written to the storage medium. The structure of the segments used within the physical log is shown in Figure 3.



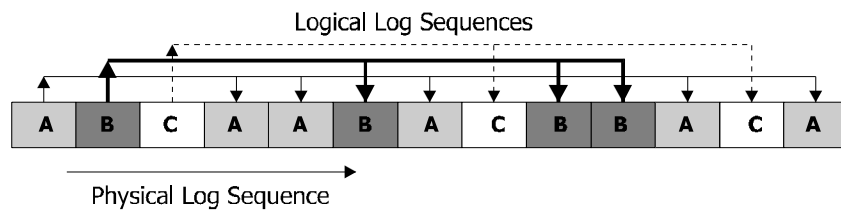
**Figure 3:** Structure of segments within the physical log.

Each segment is divided into two sections, the smaller of which is used by the log server, the other by its clients. The client section of a segment is not used by the log server and its structure is completely arbitrary. The section used by the log server contains meta-data used to maintain the physical log and is known as the *segment trailer*. The fields within the segment trailer serve three purposes:

- Forming the sequence of segments within the physical log.
- Identification of the logical log to which a segment belongs.
- Finding the end of the log when the system recovers after a crash.

The *forward* and *backward link* fields hold the identity of the next and previous segments in the physical log sequence. This allows the physical log to be traversed in either direction and reduces the complexity of removing a particular segment from the log. The *log ID* field holds the identity of the logical log to which the segment belongs, and the *time-stamp* field contains the time at which the segment was appended to the tail of the log. These fields are filled in by the log server just before each segment is written to secondary storage. Thus, any information a client places in the trailer area will be lost.

Logical logs are implemented as an implicit sub-sequence of the physical log by way of the *log ID* field within the segment trailer. As shown in Figure 4, the sequence order of each logical log follows that of the physical log through which it is woven.



**Figure 4:** Three logical logs implemented as implicit sub-sequences within the physical log.

### 3.2 Management of the Physical Log

Conceptually, management of the physical log is relatively straightforward. The log server maintains a record of the identity of the segments at the head and tail of the physical log and a map of all segments to keep track of free space. When a client writes a segment to a logical log, the log server fills in the trailer information to record the identity of the logical log, allocates a free segment using the segment map, and writes the data to secondary storage.

From time to time, failures occur which require the log server to be restarted. On such occasions, the internal state must be recovered before service can resume. If the location of the head of the physical log is known, the state of the segment map can be reconstructed from scratch. This is achieved by traversing the log from head to tail, marking each of the segments encountered as used. Unfortunately, this recovery method suffers from potentially lengthy delays depending on the number of segments comprising the physical log. To speed up the recovery process, the log server can preserve its internal state by writing the free map and the identity of the segment at the end of the log to a fixed checkpoint region on the storage medium. To guard against I/O failures during such an operation, two such regions are used in strict alternation according to Challis' algorithm [4]. In case of failure, only those segments written after the last recorded tail of the log need be scanned.

To detect when the end of the log has been reached on recovery, the *time-stamp* field within the segment trailer is checked to ensure that it is greater than that of the previous segment. When a segment containing a lower time-stamp is encountered, the end of the log has been reached. This scheme works reliably provided that the contents of each *time-stamp* field is part of a previously written segment and is not simply a random pattern

of bits from an untouched region of the storage medium. To guard against this, all segment trailers must initially be filled with zeros before the storage medium can be used by the log server.

## 4. *Lumberjack*: A Hybrid Architecture

The *Lumberjack* store is a four-layer architecture comprising an *interface* layer, an *object-storage* layer, a *cache* layer, and a *log-storage* layer. These layers draw together aspects of the systems described in sections 2 and 3. In particular, the log-storage layer is based on the *log server* used in the Grasshopper system and is responsible for the low-level management of the log. The object-storage layer employs an adapted version of the object-addressing scheme used in the CPOMS and is responsible for the storage and retrieval of objects in the log. The cache layer is used as an interface between the *object-* and *log-storage* layers.

The interface layer is used to implement the store functionality required by an application system and is not specific to *Lumberjack*. For example, it may be used to provide a fully transactional object store or something as simple as a single-client checkpoint-style store. For simplicity, the remainder of this paper assumes that the store is supporting a single-client PS-algol or Napier88 store. Supporting a more complex multi-user transactional store would not require any changes to be made to the cache or log-storage layers. Changes would be required to the object storage layer to support the additional complexity required in the snapshot and recovery mechanisms. In particular, extra meta-data must be written to identify clients and transactions. These changes do not fundamentally alter the design of the store. Due to space limitations, the *interface* layer is not described in this paper. However, the design and implementation of the remaining layers are described in sections 4.1 to 4.3.

### 4.1 Log-Storage Layer

The log-storage layer in *Lumberjack* is provided by an implementation of the log server described in Section 3. Since the *Lumberjack* store is intended for use with conventional operating systems, the log server uses the file system to provide secondary storage. The current implementation uses a single file to hold the physical log although, on systems such as Unix, it is possible to store the log on a raw device partition.

INTERFACE	LOG-STORAGE LAYER
<pre>create_log(Out log_id) destroy_log(In log_id) write_segment(In log_id, In buffer, Out seg_id) read_segment(In log_id, In seg_id, Out buffer) read_first(In log_id, Out seg_id, Out buffer) read_next(In log_id, InOut seg_id, Out buffer) free_segment(In log_id, In seg_id) synch()</pre>	

Most of the operations in the interface to the log server require the identity of a previously created logical log. Initially, a client will use the *create\_log* operation to obtain access to an empty logical log. The operation returns the identity of the new log in the form of a *password capability* [12] to prevent forgery. When a log is no longer required, the *destroy\_log* operation may be used to dispose of it.

Data may be stored in a logical log using the *write\_segment* operation, which is supplied with a buffer containing the contents of the segment to be appended to log. The log server fills in the segment trailer information and writes the buffer to an empty segment on secondary storage. The identity of the segment is returned to the caller so that the data may subsequently be retrieved. It is important to note that the log-storage layer is not permitted to employ any form of write-back caching strategy, which ensures that the *write\_segment* operation is performed synchronously.

Three operations are provided to retrieve segments previously written to a logical log. If the identity of the segment to be read is known, the *read\_segment* operation can be used to load the contents of the segment into the buffer provided. The log server checks that the requested segment lies within the caller's logical log and if not, the operation fails. During crash recovery, the *read\_first* and *read\_next* operations may be used to traverse the sequence of segments in a logical log. The *read\_first* operation is used to retrieve the first segment in a logical log. The identity of this segment is returned for use in retrieving the remaining segments. This is achieved using the *read\_next* operation, which uses the identity of the current segment to retrieve the next segment in the logical sequence. As before, the identity of this segment is returned so that the traversal of the logical log may be continued.

The *free\_segment* operation is provided to inform the log server that a segment no longer contains any useful data. Such segments may be reclaimed by the log server and used to fulfil subsequent *write\_segment*

operations. The final operation, *synch*, is used to force the log server to preserve its internal state as described in section 3.2. This typically occurs when the store is being shutdown or to minimise the cost of recovery. Importantly, it is not necessary for the *synch* operation to be invoked each time a user process performs a snapshot or commit operation on the store. This aspect of the store is discussed in more detail in section 4.3.6.

## 4.2 Cache Layer

The *cache* layer acts as an intermediary between the *object-* and *log-storage* layers. It uses the interface provided by the *log-storage* layer and exports a simpler high-level interface for use by the *object-storage* layer. Its purpose is to minimise the number of log I/O operations required by caching recently used segments in main memory. From a conceptual viewpoint, the cache contains a number of *slots*, each capable of holding a single segment. The contents of a particular slot may have been loaded from the log, or it may contain a new segment that is being filled with data. If a segment held in a slot has been assigned an identity (through being written to the log), the identity is held in the meta-data associated with the cache slot.

INTERFACE	CACHE LAYER
<code>new_segment(Out slot_id)</code>	
<code>flush(In slot_id)</code>	
<code>find_segment(In seg_id, Out slot_id)</code>	

The cache layer supports three operations as shown above. The *new\_segment* operation is used to allocate a cache slot containing an empty segment that may subsequently be filled with data. Once a segment is full, the *flush* operation may be used to append the segment to a logical log. It should be noted that this operation does not cause the segment to be removed from the cache. It is merely written to the log and assigned a segment identifier, which is subsequently stored in the meta-data associated with the cache slot.

The *find\_segment* operation is used to access an existing segment given its identity. If the segment is already resident in the cache, the operation simply returns the identity of the cache slot in which it may be found. Otherwise, a cache slot is allocated and the requested segment is loaded from the log. To support the *find\_segment* operation, it must be possible to determine whether a particular segment is loaded. Thus, a data structure mapping the identity of a loaded segment to a particular cache slot is required. Since segments created as a result of the *new\_segment* operation do not have an identity, they will not be entered in the cache map until they are written to the log. Until this time, data held in such segments must be located via alternative means as described in section 4.3.2.

## 4.3 Object-Storage Layer

The object-storage layer is implemented using the cache layer and introduces the abstraction of an *object* to the upper layers. The object-storage layer has the following responsibilities:

- Managing the storage of objects within a logical log.
- Recovery of objects following a crash.
- Garbage collection and re-clustering of objects within the log.

The following sections describe how each of these responsibilities is carried out.

### 4.3.1 Object Format

The object-storage layer defines the layout of the objects it manages to enable the provision of garbage collection, object re-clustering, and log cleaning services. In order to implement these services, it must be possible to locate pointers from one object to another. Since the initial implementation of *Lumberjack* is intended to support the *Persistent Abstract Machine* (PAM) layer of the *Napier88* system, the object format exported by the object-storage layer is the same as that used by the PAM. Although the current implementation uses the same object format as the PAM, this is not a requirement of the store. It would be a simple matter to change the object format to suit the requirements of an arbitrary application system.

Each object managed by the object-storage layer is assigned a unique *persistent identifier* (or PID) that may be used to refer to the object. A PID is simply a flat bit pattern and has no internal structure. As shown in Figure 5, each object contains two sections: a *pointer* section and a *non-pointer* section. Each of these sections is composed of zero or more 32-bit words. The first word of an object (word 0) specifies the number of words in the *pointer* section, and the second word (word 1) specifies the number of words in the entire object. The *pointer*

section of an object is used to hold the PIDs of other objects in a manner enabling them to be easily located by a garbage collector or other such utility.

Number of pointers	Size	Pointers	Non-Pointers
--------------------	------	----------	--------------

**Figure 5:** Format of objects supported by the object-storage layer (and PAM).

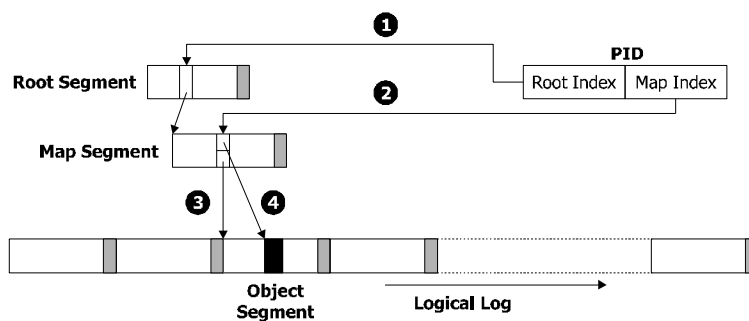
The object-storage layer must distinguish between three different states of objects: *volatile*, *stable*, and *recoverable*. Volatile objects are those that have just been created and those that have been modified since being loaded from the log. It is assumed that the state of these objects will be lost on failure. Stable objects are those that have been written to the log, and hence will survive a failure. However, objects are only recoverable once they are stable and part of a valid snapshot.

### 4.3.2 Locating Stored Objects

The object-storage layer maintains two separate data structures to record the locations of objects. The first of these data structures maps a PID to an address within a logical log and it is used to locate *recoverable* objects. This data structure is known as the PID to LOG map (or PIDLOG) and it bears much resemblance to the map used for external addressing in the CPOMS. However, the CPOMS map is required to be updated synchronously at the time of a snapshot. In contrast, the PIDLOG may be written to disk asynchronously due to the applicative nature of the store and the fact that segments are self-describing. To facilitate the recovery process, the entries in the PIDLOG are stored in segments held in a separate logical log.

The segments comprising the PIDLOG form a two-level hierarchical map that is described by a single *root segment*, the entries of which refer to *map segments* held within the same logical log as the root. These map segments form the second level of the hierarchy and they are used to locate objects stored within segments in a separate logical log. The hierarchical structure of the PIDLOG is indexed by two implicit fields within the PID of each object. In effect, a PID is used in much the same as a virtual address is used to index a page table.

To locate a particular object, an entry within the *root segment* is selected using the *root index* field from the PID, as illustrated in step one of Figure 6. This entry contains the identity of a second-level *map segment*, which must be loaded into the cache if it is not already resident. In step two, the *map index* field from the PID is used to select an entry within the previously loaded map segment that contains two pieces of information. First, the entry identifies a particular segment in which the object we require is stored. This information is used to load the segment into the cache, thereby allowing access to the objects it contains (step three). The second piece of information is the offset of the required object within the segment. This allows the object data to be copied out of the cache and passed to the upper layer for processing (step four).



**Figure 6:** Using the PID to LOG map to locate a stored object.

The second data structure is known as the *Active Object Map* (AOM) and it is used to locate the current versions of *volatile* and *stable* objects. Since the current versions of volatile objects are held in the cache, the AOM is capable of mapping a PID to a particular cache slot and an offset within that slot. When a segment containing volatile objects is evicted from the cache and written to the log, the objects become stable, but are not recoverable until a snapshot is requested by the upper layer. The upshot of this is that the locations of the stable objects within the log cannot be recorded in the PIDLOG and must be managed elsewhere. Thus, the AOM is also able to map from a PID to a location in the log.

Whilst an object remains in the cache its location is described by a PID to CACHE entry in the AOM. This entry is changed to a PID to LOG entry when the object becomes stable and is no longer resident in the cache. Eventually the upper level will perform a snapshot operation causing all volatile and stable objects to become recoverable. At this time, the entries in the AOM are used to update the PIDLOG and are subsequently purged.

Since the AOM does not contain any information about the location of recoverable objects, its contents are of no value following a crash. Thus, the AOM is a transient data structure.

During the operation of the store, requests to access particular objects will arrive from the upper layers. The AOM and PIDLOG maps are used to resolve such requests in the following manner. Given a PID, the AOM is first consulted to locate a *volatile* or *stable* version of the object. If this succeeds, the upper layer is passed a copy of the object data from the location indicated by the AOM. On the other hand, if the AOM contains no information about the required object, the PIDLOG is consulted to find the most recent recoverable version of the object.

### 4.3.3 Storing Objects in a Logical Log

The principle task of the object-storage layer is to manage the storage of objects within a logical log. Within a logical log, objects are stored in segments, which leaves two questions to be answered:

- Under what circumstances is an object stored within a segment?
- When these circumstances arise, how is the object data structured within a segment?

In answer to the first question, an object is stored in a segment when the interface layer creates a new object. Once an object has been stored in a segment, it may not subsequently be changed. Instead, updates are made by storing a new copy of the object, thereby rendering the previous version obsolete. Thus, an object is also stored in a segment when an updated version of an existing object is created.

As new and updated versions of objects are created, the object-storage layer packs the object data from “left” to “right” within an empty segment provided by the cache layer. As each object is stored, a small amount of meta-data is introduced to ensure that the contents of the segment are self-describing. In particular, the data for each object is preceded by a single word containing its PID. Thus, by using the *size* field within the object data, a linear scan may subsequently be used to locate all of the objects within a segment.

As described in section 4.3.6, it is also necessary to store *meta-records* relating to the management of the store. Such records are interspersed with the storage of objects and must be independently identifiable. The current implementation represents each meta-record using a reserved PID followed by a *type* indicator. The reserved PID is used to distinguish meta-records from stored objects and the *type* indicator is used to identify the meta-record. Presently, three kinds of meta-record are used: one to signal the occurrence of a *snapshot* operation, one to mark the start of object fragments that have overflowed a segment boundary, and the other to record the fact that the store has recovered following a failure.

### 4.3.4 Cache Eviction Upcalls

As the object-storage layer populates segments with object data, the cache will fill with segments ready to be written to the log. Eventually, the cache layer will need to evict one of these segments to make room for the storage of new objects. When this occurs, the cache layer will select a cache slot and write the corresponding segment to the log, thereby causing the objects within the segment to become stable. Since it is necessary to keep track of the location of stable objects, the cache layer performs an *upcall* to the object-storage layer after the segment has been written to the log, but before it is evicted from the cache. This provides an opportunity for the object-storage layer to scan the contents of the segment and update the AOM entry for each object.

Cache eviction upcalls are performed whenever any segment is evicted from the cache. Most of the time, these calls will result from the eviction of a segment containing object data. However, occasionally they will be made when a modified PIDLOG map segment is evicted. When this occurs, the location of the map segment must be updated in the root segment to enable the object addressing mechanism to function correctly.

### 4.3.5 Snapshot

From time to time, the application system supported by the store will issue a *snapshot* request to ensure that all previous updates to objects are recoverable. Snapshot requests are handled by writing all of the segments containing volatile object data to the log using the *flush* operation provided by the cache layer. Following this, the PIDLOG must be updated to record the location of the newly recoverable objects within the log. This is achieved in two stages; firstly, the AOM is scanned to locate all entries referring to objects that were already stable at the time of the snapshot. Such entries are copied to the PIDLOG and are then removed from the AOM. Secondly, the segments written to the log using the *flush* operation are scanned to locate the stored objects. As each object is found, the PIDLOG is updated to record the location of the recoverable version of the object.

An important step that has been omitted from the above description is the logging of a *snapshot* meta-record that is used during recovery to determine which objects are recoverable. This meta-record is simply packed into the first available space in a cached segment in the same manner as object data. It is important that the snapshot meta-record be logged before the volatile data is flushed from the cache. Otherwise, a failure

occurring after the snapshot, but before the meta-record is written to the log, will cause all of the updates since the last snapshot to be discarded.

When a snapshot is performed, the PIDLOG map segments that describe the location of the newly recoverable objects are updated, but not written to the log. Since the segments containing the recoverable object data are self-describing, it is not necessary to write the modified map segments to the log eagerly. Instead, they may be written back lazily as policy dictates, or perhaps not at all. Writing the map segments to the log eagerly results in short recovery times but increases the overhead of each snapshot operation. On the other hand, not writing the map segments to the log at all will allow for very efficient snapshots but will result in lengthy recovery times. Thus, there is a trade-off between the speed of recovery and the overhead of a snapshot.

#### 4.3.6 Recovery

In any persistent store, it is necessary to recover from failures to ensure the resilience of the stored data. In the *Lumberjack* store, the recovery procedure allows for a remarkable degree of flexibility due to a trade-off between the overhead of snapshot operations and the speed of recovery. For example, if failures are infrequent, then it is possible to employ an *optimistic* strategy in which there is very little overhead imposed during snapshots. The cost of this is the potential of lengthy recovery times. Alternatively, a *pessimistic* strategy may be adopted in which rapid recovery can be guaranteed at the cost of an increase in the overhead of snapshot operations. These two strategies represent opposite ends of a spectrum and in practice, policy may be tuned to achieve a middle ground such as the writing back of meta-data when the I/O system is quiescent.

In the *Lumberjack* store, the recovery procedure is conceptually very simple. In the worst case, in which no PIDLOG map segments have been written to the log, recovery is performed by scanning every segment in the log from head to tail. During this scan, the recovery procedure is interested in locating the following things:

- Object data.
- *Snapshot* meta-records.
- *Recovery* meta-records.
- The end of the log.

Initially, all objects found in the log must be treated as stable until a *snapshot* meta-record is encountered. Thus, as each object is found, an entry is made in the AOM to record the stable location of the data. Under normal circumstances, a *snapshot* meta-record will eventually be found. When this occurs, the entries in the AOM are used to recreate the PIDLOG and are then purged. Any entries remaining in the AOM when the end of the log is reached refer to objects that are not recoverable since they are not part of a snapshot. Thus, when the end of the log is reached, the AOM is cleared in preparation for the resumption of normal operation.

Before the store can resume operation, it is necessary to record the fact that a failure has occurred. This is achieved by creating a new segment in the cache and logging a *recovery* meta-record. The purpose of this meta-record is to prevent any non-recoverable objects discovered during the present recovery procedure from being mistakenly recovered in the future. Therefore, if a *recovery* meta-record is encountered during the scanning of the log, the contents of the AOM must be cleared.

The procedure outlined above is only necessary in the worst case. Ideally, recovery will be facilitated by locating the most recent version of the root segment of the PIDLOG in the log. This root segment describes the location of all recoverable objects up until its position in the log. Using the procedure outlined above, it is possible to *roll* the PIDLOG *forward* by scanning the segments written to the log after the root segment. Thus, the frequency with which the root segment is written to the log directly influences the time required for recovery.

The location of the most recent version of the root segment is recorded in the root block used by the log-storage layer to describe its checkpoint region. Thus, whenever the root segment is written to the log, it is necessary to update its location in the root block. Again, this may occur eagerly or lazily depending on the particular policy in use.

One issue that has yet to be discussed is the recovery of the PIDLOG map segments. Since all segments are held within the same *physical* log, the recovery procedure outlined above will encounter both segments containing object data and segments that are part of the PIDLOG map. The data stored in map segments has a different structure to that within object segments. Therefore, unless map segments can be distinguished from object segments, the recovery procedure will misinterpret the contents of any map segments it encounters. To guard against this, map segments are written to a separate logical log allowing them to be skipped by the recovery procedure.

### 4.3.7 Garbage Collection, Re-clustering, and Cleaning

Once the store has been operating for some time, the log will fill with obsolete versions of objects and map segments. At the same time, the application system supported by the store will generate *garbage* in the form of objects that are no longer reachable from any persistent root. Obsolete and unreachable objects along with old versions of map segments are wasteful of space in the log and must be eliminated.

In a typical log-structured store, a *cleaner* is periodically run to reclaim space occupied by segments that contain obsolete data. This is achieved by copying the *live* data out of these segments and re-clustering it within new segments that are eventually written to the tail of the log. The cleaner behaves in exactly the same way as a store client in that it non-destructively modifies the live objects in the store and performs snapshot and *free\_segment* operations. A necessary requirement of any cleaner is the ability to decide which data is *live* and which is obsolete. In general, this is achieved by keeping track of the location of the most recent version of each object. This information can then be used to decide whether a particular version of an object is the most recent version, and therefore whether or not the object is live. In the *Lumberjack* store, the cleaner is able to consult the AOM and PIDLOG to decide whether a particular version of an object is live. Since segments are self-describing and store writes are applicative, this may be performed incrementally and safely in parallel with store computation.

Most application systems that employ *persistence-by-reachability* include a facility to perform garbage collection of unreachable objects. Simply put, the purpose of the garbage collector is to locate objects that are no longer required and destroy them so that they do not occupy space in the persistent store. When an application-level garbage collector is supported by a log-structured store, the log cleaner can be used to eliminate the garbage objects whilst it is reclaiming the space occupied by obsolete data. However, to exploit the actions of the cleaner in this way, there must be an exchange of information between the cleaner and the garbage collector. In other words, the cleaner must be made aware of which live objects are deemed to be garbage by the application system. The manner in which this is achieved is the subject of current research and the *Lumberjack* store is intended to provide a suitable platform for experiments in this area.

## 5. Conclusion

In this paper, we have described a novel persistent object store architecture that exploits the benefits of log-structured storage techniques. The benefits of this architecture are:

- The amount of meta-data that must be written at the time of a checkpoint or transaction commit is minimised, thus potentially providing fast snapshots and commits.
- Writing to the log always involves writing to a physically contiguous region of disk which amortises seek time.
- The time at which meta-data is written is flexible providing a great deal of scope for implementing policy. For example, meta-data may be written during quiescent periods permitting disk systems to be efficiently utilised.
- The store is amenable to re-clustering by processes that are identical to ordinary clients of the store.
- The store is amenable to, and we believe provides an ideal substrate for, incremental parallel store garbage collection techniques, for example, PMOS [11].

The initial implementation of *Lumberjack* is proceeding in parallel with the writing of this paper and serves a number of purposes:

- It delivers object-based logging for use on conventional operating systems and is intended to support applications such as *Napier88* [9] and persistent Java [2].
- It provides a platform that enables empirical comparisons to be made between log-structured store designs and the various alternatives. This is significant because the use of log-structured storage is relatively new in the field of persistent systems and it would be beneficial to carry out a performance study.
- It provides a suitable framework to conduct experiments relating to the interaction between garbage collection and the management of free space in the log.

At the present time, the lower layers of the architecture are complete and work is proceeding on interfacing the store with the *Persistent Abstract Machine* interpreter used to support *Napier88*. It is hoped that the final version of this paper will contain some preliminary measurements of the performance of the store.

## References

- [1] M.P. Atkinson, K. Chisholm, and W. Cockshott, "PS-algol: An Algol with a Persistent Heap", *ACM SIGPLAN Notices*, **17**(7), pp. 24-31, 1981.
- [2] M.P. Atkinson, L. Daynes, M.J. Jordan, T. Printezis, and S. Spence, "An Orthogonally Persistent Java<sup>TM</sup>", *ACM SIGMOD Record*, **25**(4), pp. 1-10, 1996.
- [3] A.L. Brown and W.P. Cockshott, "The CPOMS Persistent Object Management System", *Technical Report PPRR-13*, Universities of Glasgow and St Andrews, 1985.
- [4] M.F. Challis, *Database Consistency and Integrity in a Multi-User Environment*, in *Databases: Improving Useability and Responsiveness*. 1978, Academic Press. p. 245-270.
- [5] K. Elhardt and R. Bayer, "A Database Cache for High Performance and Fast Restart in Database Systems", *ACM Transactions on Database Systems*, **9**(4), pp. 503-525, 1984.
- [6] D. Hulse, "Store Architecture in a Persistent Operating System", *Ph.D.*, Department of Computer Science, University of Adelaide, Adelaide, 1998.
- [7] D. Hulse and A. Dearle. "A Log-Structured Persistent Store", in *Proceedings of the 19th Australasian Computer Science Conference*, University of Melbourne, Melbourne, pp. 563-572, 1996.
- [8] C. Mohan, D. Haderie, B. Lindsay, H. Pirahesh, and P. Schwarz, "ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging", *Research Report RJ 6650*, IBM, 1989.
- [9] R. Morrison, A.L. Brown, R.C.H. Connor, and A. Dearle, "The Napier88 Reference Manual", *Technical Report PPRR-77-89*, University of St. Andrews, St. Andrews, 1989.
- [10] J.E.B. Moss, "Working With Persistent Objects: To Swizzle or Not to Swizzle", *Technical Report 90-38*, University of Massachusetts, Massachusetts, 1991.
- [11] J.E.B. Moss, D.S. Munro, and R.L. Hudson. "PMOS: A Complete and Coarse-Grained Incremental Garbage Collector for Persistent Object Stores", in *Proceedings of Proceedings of the Seventh International Workshop on Persistent Object Systems*, Cape May, New Jersey, pp. 140-150, 1996.
- [12] S.J. Mullender, G. van Rossum, A.S. Tanenbaum, R. van Renesse, and H. van Staveren, "Amoeba: A Distributed Operating System for the 1990s", *Computer*, **23**(5), pp. 44-53, 1990.
- [13] J. Rosenberg, A. Dearle, D. Hulse, A. Lindström, and S. Norris, "Operating System Support for Persistent and Recoverable Computations", *Communications of the ACM*, **39**(9), pp. 62-69, 1996.
- [14] M. Rosenblum and J.K. Ousterhout. "The Design and Implementation of a Log-Structured File System", in *Proceedings of the 13th ACM Symposium on Operating System Principles*, Pacific-Grove, California, pp. 1-15, 1991.
- [15] S.J.G. Scheuerl, R.C.H. Connor, R. Morrison, and D.S. Munro. "The DataSafe Failure Recovery Mechanism in the Flask Architecture", in *Proceedings of the Nineteenth Australasian Computer Science Conference*, Melbourne, Australia, pp. 573-581, 1996.