# A Log-Structured Persistent Store

*David Hulse*

Department of Computer Science
University of Adelaide
S.A., 5005, Australia

*dave@cs.adelaide.edu.au*

*Alan Dearle*

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

*al@cs.stir.ac.uk*

## Abstract

*Persistent stores have been implemented using a variety of storage technologies including shadow paging, log-based and log-structured approaches. Here we compare these approaches and advocate the use of log-structuring. The advantages of such a technique include efficient support for large (64 bit) address spaces, scalability and fast snapshot processing. We describe the architecture of a new log-structured persistent store and how it has been used to support resilient persistent processes in the context of the Grasshopper operating system. This store is based on the use of a log server which provides clients with private logical logs.*

**Keywords** persistent stores, log-structured stores, recoverable virtual memory.

## 1 Introduction

The concept of persistence may be defined as the attribute of data which specifies its period of existence. Stemming from this simple definition, a wide range of systems supporting various forms of persistence have emerged. In systems that support *orthogonal persistence* [1], the application programmer does not have to manage the movement of data to and from backing store nor translate between different data formats. Most of these systems are supported at the lowest level by a *persistent store* which provides a stable and resilient data repository.

This paper describes a new persistent storage architecture based on *logging*. This store is designed to support paged persistent address spaces in which processes directly execute – i.e. it supports persistent virtual memory. It differs from other persistent storage architectures in that it:

- is designed to support large (64 bit) address spaces,
- is designed to support large or small data sets (i.e. it is scalable), and
- is designed to support frequent fine-grained snapshots.

The architecture has been implemented and is being used to support persistent *containers* in the Grasshopper operating system [6].

The next section describes various persistent store technologies, highlights their strengths and weaknesses and justifies our design decisions. Section 3 describes a central part of the architecture, known as the *Log Server*, which provide clients with an abstract interface to logical logs. Section 4 discusses how these logs are used by clients to implement persistent stores. Section 5 describes how the store is used in the context of the Grasshopper persistent operating system. Section 6 concludes.

## 2 Persistent Store Technologies

Researchers have experimented with many different approaches to the implementation of persistent stores. These include shadow paging techniques, log-based approaches, and more recently, log-structured approaches [12, 8, 14]. This section introduces these techniques, highlights the differences between them and justifies why we have decided to design and implement a new log-structured architecture.

The approaches listed above can all support a persistent virtual address space capable of moving atomically from one state to another – an action known as *checkpointing*, *snapshotting* or *stabilising* the store. Atomicity is important since the integrity of the store must never be jeopardised in the event of a system crash.

In the remainder of this paper we shall assume that the persistent store subsumes the role of virtual memory and that all processes run in a paged persistent address space. This assumption is true in the context in which this work was carried out and somewhat simplifies arguments. However, if this assumption is relaxed, we believe that the arguments made in favour of log-structured approaches remain valid.

## 2.1 Shadow Paging

A shadow paged virtual memory system is similar in operation to a standard virtual memory system in that both systems control the movement of pages to and from physical memory. In contrast to conventional virtual memory, a shadow paged system ensures that a modified page never overwrites its original version when being written back to non-volatile storage. Provided that appropriate housekeeping information is maintained, this ensures that changes can be undone or redone in the event of failure.

Checkpoint atomicity may be achieved through the use of a mirroring technique such as Challis' algorithm [3]. Using Challis' algorithm, the persistent store is described by two timestamped root blocks which reside at known disk locations. The final action of a checkpoint is to overwrite the oldest root block with a new root block that describes the new state of the store. If this write is successful, the checkpoint is complete and a new state is established. The algorithm guards against writing a partial root block by duplicating the timestamp at the start and end of the root block. A root block is considered valid only if the timestamps at its start and end are identical. This technique is illustrated in conjunction with after look shadow paging in Figure 1. The next sections describe two commonly used forms of shadow paging known as *after look* and *before look*.

### 2.1.1 After Look Shadow Paging

In an after look shadow paging system [12, 13, 15], a dirty page is never written to the site from which the clean version of the page was fetched; this is analogous to *deferred-write logging* [4, 9]. Instead, modified pages are *shadowed* by writing them to unused disk blocks. Since the locations of pages change, it is necessary to maintain a *Logical to Physical Mapping* (LP-Map) which maps virtual addresses onto locations on non-volatile storage.

The LP-Map is a persistent data structure a copy of which is typically kept in main memory for efficiency reasons. However, larger stores require proportionately larger LP-Maps which may not fit entirely within memory. In this case the map could be paged from a transient region of disk which would require a separate mapping. This introduces complexities which some systems, such as CASPER [15], have solved by embedding the LP-Map in the persistent address space that it describes. This has the advantage that it will be paged and shadowed as it is modified using a single mapping.

Atomicity is achieved through the use of Challis' algorithm. If the embedded LP-Map technique is employed, the root block contains the disk address of the root of the LP-Map and the entire persistent virtual memory is self describing. Figure 1 shows a persistent store containing two recoverable states referred to by the two root blocks. The state of the store is initially described by root block 1 and contains four pages of data. During subsequent computation, page 3 is modified resulting in page 5 being allocated as a shadow. On snapshot, an updated LP-Map is written to page 6 and a new recoverable state is formed; this is referred to by root block 2.

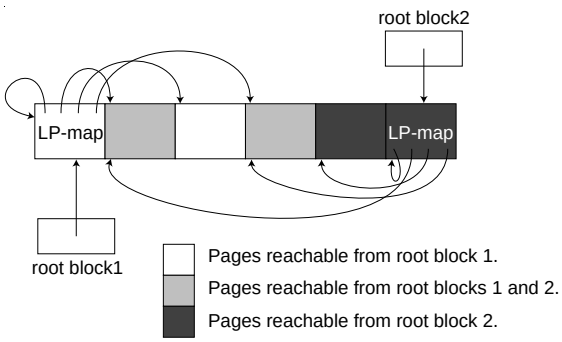


Figure 1: A persistent store containing two recoverable states.

Since the LP-Map encodes the locations of the data on non-volatile storage, it must be written whenever a checkpoint is made. If checkpoints are infrequent, the address space small, and the amount of modified data large, this overhead is not onerous. However, for large address spaces with frequent checkpoints, this overhead becomes significant. Using conventional page tables to represent the LP-Map requires meta data overheads of between 6% (dense) and 400% (sparse) for 100 pages on a 64 bit architecture [10]. Consider the worst case of a process that modifies a single page of data and checkpoints on a 64 bit architecture. With an 8K page size, if the LP-Map is represented as a conventional page table containing an 8 byte disk

address, six pages of LP-Map must be written to non-volatile storage to describe the changes made to the single page. These observations have lead others [8, 9] to conclude that the after look approach is inappropriate for use in large applications requiring high throughput.

A final disadvantage of the after look technique is that physical clustering of pages is prevented since modified data is written to unused disk blocks; this degrades read performance. However, if contiguous blocks can be found on disk, writes may be batched considerably improving checkpoint performance.

### 2.1.2 Before Look Shadow Paging

Systems based on the before look approach to shadow paging [2, 8] operate similarly to after look systems in that the shadow pages are always written to unused blocks on disk. In contrast to after look systems, the shadow pages are copies of the original pages allowing modified pages to be written to the disk locations from which they were fetched. The locations of the copied original pages must be written to disk before any modified pages to ensure that incomplete updates can be safely undone in the event of a failure. In effect, the set of shadow pages form an undo log. After a successful snapshot operation, this log is discarded and the space occupied by the shadow pages is free for reuse.

Since pages always reside at the same location on disk, the mapping of virtual addresses to physical disk locations is much simpler than with after look shadow paging. In some systems this mapping is completely static as demonstrated by Brown's before look shadow paged store [2].

Before look has the advantage that it preserves physical clustering of data on disk which can improve read performance. However, since modified data is always written to the location from which it was read, writes are prone to high seek and latency delays. The major disadvantage of the before look technique is the additional I/O required to write the shadow pages and their locations to disk before any modified pages can be written.

## 2.2 Log-based Stores

An alternative approach to shadow paging is to make use of the properties of logging to overcome some of the difficulties encountered with either form of shadow paging. The primary advantage of logging is that writes are appended to the end of a log in a sequential stream. This eliminates the I/O bottleneck associated with the random writes of shadow paged systems since continuous disk access in the same or neighbouring cylinders can be faster by an order of magnitude [14].

We define a *log-based store* as one in which a log is used in conjunction with another approach such as shadow paging. We use the term *log-structured approach* to refer to a store which uses only logs for persistent storage. An effective use of a log-based approach can be seen in the design of the DB Cache [8]. This database system was designed to provide efficient throughput of transactions whilst maintaining high availability which required fast failure recovery. The DB Cache is comprised of three integral parts: a volatile cache, the physical database and a non-volatile *safe* approximately the same size as the cache.

The cache is used to hold all of the currently active pages. The physical database resides on disk and contains a single version of each page. Pages in the database are updated in place which permits the mapping from virtual addresses to disk locations to be static. A page in the database is either the current version or it is obsolete in which case the current version is held in the cache. Together, the cache and the physical database represent the actual state of the database.

During the running of the system, pages are operated on in the cache. Pages are read in from the physical database when they are first required. When a page is modified, a copy is made within the cache and modification is allowed to proceed on the copy. Snapshots are processed by logging the modified pages to the *safe* and marking the cached versions as originals but changed with respect to the physical database. When pages are evicted from the cache, they are written to their fixed location within the physical database if they are changed, otherwise they are simply discarded.

The purpose of the safe is to protect the cache against loss through system failure. It contains copies of the pages needed to reconstruct the cache following a failure. In particular, it contains those pages which have not yet been written back to the physical database. The other pages in the cache are either unmodified or are modified but awaiting snapshot and are of no importance following a failure. It is possible to reconstruct the state of the database after a crash by reading the contents of the safe into the cache and marking each page as changed with respect to the physical database.

The positive aspects of the approach taken by the DB Cache are that the use of a log makes both

snapshot and recovery efficient. In contrast to the LP-Map required by after look shadow paging, very little meta information needs to be maintained. However, the main problem with this technique is the limitation that pages being modified by a transaction cannot be swapped out. They must remain in the cache until the transaction either commits or aborts. This causes problems in a system where transactions often run for long periods of time since the cache may become full of modified pages that cannot be swapped out. This problem can be solved by logging copies of the original pages and then writing the modified versions directly into the physical database. Naturally, this introduces some of the problems associated with before look shadow paging.

A final problem with the DB Cache approach is that pages in the safe require associated meta information to describe where they are located within the physical database as well as some other flags needed for correct operation of the safe. In the system described in [8], this information is contained in the header of each page. This approach is unsuitable for use in a page based persistent store.

## 2.3 Log-structured Stores

In a log-structured store all data is stored in a log comprising a conceptually infinite sequence of records. Records in the log are never overwritten – i.e. logs exhibit append-only semantics. As we shall see, this append-only behaviour can be exploited to efficiently support persistent stores.

The mapping of a persistent address space onto a log may be achieved by writing modified pages and meta data describing the virtual address of those pages into the log. The log therefore contains a serial history of the changes made to the store. The basic mechanism of the log may be trivially extended to support the ability to move atomically from one state to another. This is achieved by writing a special snapshot meta record into the log signifying that a new consistent state has been established. On restart, a consistent state may be found by reading only those records written before the last snapshot meta record. Since records occurring after the last snapshot meta record are not part of a consistent state, care must be taken to distinguish them from newly written records following a restart. This may be handled by either writing a restart meta record to the log or via log truncation. During execution, the location of pages in the log may be cached in a data structure similar to an LP-Map. This data structure need only be transient since it can be reconstructed by reading the consistent meta data from the log.

The physical volumes on which logs are written are bounded in size; therefore some mechanism must be provided to make space for new records to be written to the log. In practice two alternative techniques, *compaction* and *threading*, may be used to manage free space.

In a store that employs compaction, records are compacted towards the start of the log thus eliminating fragmentation and making space available at the tail of the log. The cost of compaction is that data is repeatedly moved towards the head of the log. The amount of copying performed by the system may be considerably reduced by judicious use of threading.

In a threaded store, the log is threaded through the free space on disk. However, to prevent severe fragmentation, the disk may be partitioned into fixed size *segments*. Only segments may be written to the log and when a new segment is appended, it is written in its entirety. The size of the segments is chosen such that the seek and latency time needed to start writing the segment is negligible compared to the time taken to actually write the set of contiguous disk blocks. When space needs to be recovered, one or more segments are read into memory and the live records are copied into new segments which are appended to the log. The old segments are then free for reuse.

### 2.3.1 The Advantages of Log Structuring

The removal of the requirement to maintain a persistent LP-Map coupled with a small, fixed meta-data overhead is a major advantage of the log-structured approach. This manifests itself as increased snapshot performance resulting from two factors:

- less meta data needs to be written, and
- writes are to contiguous disk regions which dramatically increases throughput.

A further benefit of the log-structured approach is that data may be dynamically re-clustered to improve physical locality by copying data to the end of the log. This gives equivalent read performance to the before look shadow paging technique and no worse performance than after look.

The implementation of a log-structured system may be usefully partitioned into general logging facilities and use of these facilities by clients. In the system described in this paper, a log server is

provided that manages a physical log and presents each client with its own logical log; the log server is described in the next section. The way in which clients use the log server to implement a persistent storage facility is described in Section 4.

## 3 The Log Server

The primary purpose of the log server is to provide an abstract interface to a logging subsystem. The log server divides its available storage into fixed size segments. The end of each segment contains a trailer record used exclusively by the log server to maintain a single physical log. The remainder of the segment is unused and is free to be filled in according to the needs of individual clients.

### 3.1 Physical Log Structure

Figure 2 shows the structure of a segment and its trailer record. The most important field in the trailer is the *Next Segment* field which holds the *segment ID* of the next segment in the physical log. A segment ID encodes a segment's disk address and is meaningful only to the log server. Since a pointer is used to identify the next segment, the log is not required to be physically contiguous. Such logs are termed *threaded* and may be likened to a singly linked list. In this case, the physical log is forward chained which simplifies scanning.

<table>
<tr><td rowspan="4">Unused by log server</td><td>Segment Trailer</td></tr>
<tr><td>Next Segment</td></tr>
<tr><td>Client</td></tr>
<tr><td>Timestamp</td></tr>
</table>

Figure 2: Segment trailer record used by the log server.

The system described in this paper provides multiple logical logs constructed above a single physical log. A logical log is a resource allocated to a single client and comprises the set of segments used by that client. A physical log is the concrete storage containing segments potentially from many clients.

The *Client* field is used to partition the physical log providing each client with a private logical log. Each logical log comprises segments with identical *Client* fields. Clients may read only those segments that they own. To support this, each client must register with the log server prior to using any of its services. At this time, the log server issues it with a unique identifier which must be presented when performing operations. When writing a segment, the log server fills in the *Client* field with the identifier supplied by the client. Our implementation of the log server within Grasshopper uses the capability system [5] to make this mechanism secure.

The *Timestamp* field is filled in by the log server before the segment is written. It is used to impose a chronological order on the segments in the physical log which is useful for finding the last segment on restart. Assuming that the log server zeroes all segment trailers before first use, the end of the log can be found by scanning the sequence of segments until the timestamp ceases to increase.

### 3.2 Log Server Interface

The log server provides registered clients with seven operations. When a client wishes to append a segment to its logical log, it calls the *write_segment* operation which has the following signature:

```
write_segment(ClientID id, Buf buffer)
              returns SegID
```

The client supplies its registration identifier and the segment it wishes to have appended to its log. The log server fills in the *Client* field of the segment's trailer record and writes the current time into the *Timestamp* field. The *Next Segment* field is initialised with the identity of an eagerly allocated unused segment. Finally, the log server enqueues the segment for writing to non-volatile storage and returns its identifier.

When the *write_segment* operation returns, no guarantees regarding the stability of the data are made. The segment may have been cached by a device controller or queued for writing by the log server. A client may force segments to non-volatile storage using the *flush_segment* operation, which causes all segments up to and including *segid* to be written.

```
flush_segment(ClientID id, SegID segid)
```

Segments can be retrieved from a client's logical log using the *read_segment* operation which has the following signature:

```
read_segment(ClientID id, SegID segid,
             Buf *buffer)
```

The client must provide the log server with its registration identifier and the identity of the segment it wishes to read. The log server checks that the client owns the specified segment; if it

does, the segment is copied into the *buffer* supplied by the client, otherwise an error is indicated.

When a client of the log server restarts following a failure, it may be necessary to scan the logical log to recover as much state as possible. In support of this, the log server provides two operations which enable a client to read the first and following segments in its logical log. The signatures of these operations are shown below.

```
read_first(ClientID id, Buf *buffer)
          returns SegID

read_next(ClientID id, SegID segid,
         Buf *buffer) returns SegID
```

The *read_first* operation requires the client to supply its registration identifier and a buffer into which the log server will copy the first segment of the client's logical log. The log server returns the identity of the copied segment which can be used to perform a *read_next* operation. Given the identity of an arbitrary segment in a logical log, the *read_next* operation may be used to retrieve the next segment in that logical log.

When a segment has been cleaned by a client and no longer contains valid data, it may be returned to the log server for reuse using the *free_segment* operation:

```
free_segment(ClientID id, SegID segid)
```

The final operation provided by the log server is *read_block* which, like the Unix *read* system call, permits an arbitrary region of a segment to be read:

```
read_block(ClientID id, SegID segid,
          Offset offset, Length len,
          Buf *buffer)
```

In the next section we will show how these facilities may be used to provide a paged persistent address space.

# 4 Storage Managers

This section describes the architecture and operation of our log-structured persistent store. We assume that each client of the Log Server implements a *manager* which supports a distinct persistent store. First we describe how segments are used to store persistent data. Next we describe the infrastructure maintained by managers to track persistent data stored in the log. Finally we show how these mechanisms are used to support the operations required by a page based persistent store.

## 4.1 Logical Log Structure

Each persistent store is completely described by the contents of a logical log. The logical log comprises a sequence of fixed size segments as described in Section 3.1. Each client imposes structure on these segments so that the data within them may be interpreted; this structure is described below.

### 4.1.1 Segment Layout

Since each manager supports a paged persistent virtual address space, the primary storage requirement is page sized records. Therefore managers structure segments as a collection of data pages and meta records describing those pages. Meta records describe why a page was placed in the log and include other information such as the page's virtual address. Meta records are allocated sequentially from the end of the segment while the pages they describe are allocated from the beginning as shown in Figure 3.

Figure 3: Internal Structure of a segment

As shown in Figure 4, meta records are of variable size and are self describing via a fixed size trailer containing *type* and *size* fields. Store managers use seven different types of meta record which are described below.

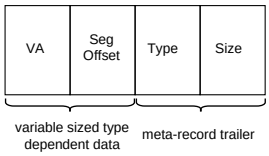


Figure 4: A *Swapped Page* meta record

- Snapshot Page: A snapshot page meta record describes a page that has been stored in the log as the result of a snapshot operation. It contains fields to record the virtual address of the page, its location within the segment and a snapshot identifier.
- Swapped Page: Swapped pages are those which have been evicted from main memory during the normal running of the system. They are not part of any snapshot and must be described by different meta records to distinguish them from other pages. The structure of a Swapped Page meta record is shown in Figure 4.

- Snapshot_Swapped_Page: When a page is swapped out and subsequently snapshotted, these meta records are used to optimise the snapshot by referring to the swapped page. Consequently, they describe the location of a page in a previously written segment.
- Snapshot_Complete: At the completion of a snapshot operation this meta record, containing a snapshot identifier, is written to signify that the snapshot is complete. This is necessary so that complete snapshots can be distinguished from incomplete ones at recovery time.
- B-Tree_Page: These meta records describe pages used to store the B-Tree nodes which are discussed in Section 4.2. They contain the virtual address of the page, an identifier for the segment in which the page is contained and the page's location within the segment.
- B-Tree_Start: Prior to writing the first B-Tree page, a B-Tree start meta record is written to the log. This guards against potential B-Tree corruption if a failure occurs when the B-Tree is being written to the log.
- B-Tree_Complete: When the last page of a B-Tree has been written to the log, this meta record is written to signify that the B-Tree is complete. Without this record, the B-Tree is invalid and an earlier version must be used.

## 4.2 Store Manager Data Structures

As in after look shadow paging systems, a manager which utilises a log must maintain data structures to enable the current versions of pages in the store to be found. Our store uses two separate data structures, the *Active Page Table* (APT) and the *Recoverable State B-Tree* (B-Tree) to describe the current location of pages in the store. Both the B-Tree and the APT reside in the virtual address space of the store manager.

The B-Tree is used to describe the current recoverable state of the store. Each node in the B-Tree describes the location of a contiguous range of pages within the logical log. This has two advantages: firstly it permits large regions of the persistent address space to be described concisely. Secondly, it efficiently describes a sparsely populated address space, which is crucial for 64 bit architectures.

The APT is used to store the current state and location of each *active* page and is purely transient. A page is said to be active if it is resident or the B-Tree does not contain an up-to-date entry for that page. Pages referenced by the APT can be either *resident* or *stable*. Resident pages are those currently stored in a physical page frame; stable pages are those which have been written to non-volatile storage. This might have occurred because the page was swapped out, or because the page was part of a snapshot. In the latter case, it is possible for a page to be both resident and stable. The location of valid pages that are not found in the APT may be found in the B-Tree; if a page is not found in either, it is invalid.

Unlike the APT, the B-Tree is a persistent data structure; however, since the meta records for the pages in the log contain virtual addresses, it is possible to rebuild the B-Tree from the contents of the logical log. Therefore the only advantage of writing the B-Tree to non-volatile storage is to increase the efficiency of restart. Consequently, the B-Tree may be snapshotted much less frequency than the data it describes.

## 4.3 Store Operations

This section describes how the logical log and meta data structures are used to implement a persistent store. For the purpose of this discussion, it is assumed that there may be one or more concurrent processes executing within the persistent virtual memory. Each will fault and snapshot independently of the others. The handling of these and other necessary operations is now discussed.

### 4.3.1 Fault Handling

When a fault occurs on a non-resident page, it is necessary to determine the current location of the page. This is achieved by consulting the APT and B-Tree, one of which stores its location in terms of a segment identifier and an offset within the segment. This information is used to perform a *read_segment* operation. Once the segment containing the page is loaded, the required page is made available and the APT is updated to reflect the fact that the page is now resident.

If the required page is already resident, the APT contains the identity of the physical page frame currently holding the page. This information is used to make the page accessible to the process. Once the page has been entered in the address space of the process it is possible to resume execution.

### 4.3.2 Snapshot

When a process executing within the persistent store performs a snapshot operation, the modified

pages it has accessed must be made stable. In our implementation of this store within Grasshopper, a list of these pages is maintained by a causal tracking subsystem [7, 11] and is supplied to the manager whenever a snapshot is made. In many systems, store managers use page protections to track these pages.

The modified pages are marshalled into segments, each described by a *Snapshot Page* meta record. As each segment becomes full, a *write_segment* operation is performed which returns the identity of the new segment. This is used in conjunction with the segment offset of each page to update the APT reflecting that the pages are now stable.

The final operation performed during a snapshot is to write a *Snapshot Complete* meta record into the log. The segments containing snapshotted pages may be written to the log server lazily and the log need not be eagerly flushed. Writing segments lazily may be used to decrease snapshot latency and increase throughput. Conversely, segments may be written eagerly using the *flush_segment* operation if required. These choices are a matter of policy.

#### 4.3.3 B-Tree Snapshot

As described in Section 4.2, B-Trees may be written to the log periodically in a lazy fashion. Before writing a B-Tree to non-volatile storage, those APT entries referring to pages that have become stable are used to update the corresponding entries in the B-Tree. Once this process is complete, updated pages containing B-Tree nodes may be written to the log and described by *B-Tree Page* meta records. These meta-records contain enough information for the virtual address space containing the B-Tree to be recreated following a restart. *B-Tree Start* and *B-Tree Complete* meta-records are used to delimit the set of *B-Tree Page* meta records. This ensures that an incomplete B-Tree snapshot can be detected should a failure occur whilst the B-Tree is being written to the log.

#### 4.3.4 Recovery

When the persistent system is restarted after a crash or after an orderly shutdown, the store manager must determine the state of the store from the contents of the log. Therefore the most recent B-Tree in the log must first be recovered; this necessitates scanning the logical log from some suitable starting point. Once the most recent B-Tree has been located, the page tables necessary to enable it to be loaded into the virtual address space of the store manager are initialised; this is possible since the *B-Tree Page* meta records contain virtual addresses.

Since B-Trees are written lazily, several snapshots might have occurred since the last time the B-Tree was stored in the log. Therefore the most recently stored B-Tree will probably not describe the most recently checkpointed state of the store. However, a description of the most recent state can be reconstructed by examining the meta records for all of the successful snapshots written after the last B-Tree. The *Snapshot Page* meta records from these snapshots can be loaded into either the B-Tree or the APT. In the system implemented, it is the APT that is brought up to date but this is largely a matter of taste. Once this is complete, the state of the store is recovered and processes may resume execution. Taken to an extreme, this technique can be used without ever writing the B-Tree to the log, as is the situation when the store is first created.

If the system has been running for some time, it is likely that the log will contain numerous versions of the B-Tree. Furthermore, since the most recent version of the B-Tree effectively summarises the snapshots earlier in the log, only the segments written after the most recent B-Tree need be scanned on restart. Therefore, the efficiency of recovery can be improved if the most recent B-Tree can be found quickly. This may be achieved by adopting the *root block* approach used in shadow paged systems to locate the B-Tree in the log. The root block stores the log address of the first meta record describing last B-Tree written to the log.

The efficiency of restart depends on the frequency with which the root block is written. However, since writing a root block adversely affects performance, the frequency with which this occurs is a matter of policy permitting a further trade off between restart and normal running costs.

#### 4.3.5 Page Eviction

When physical memory runs low, it may be necessary to swap out pages holding data from the persistent store. When a page is swapped out, its entry in the APT must be updated to record the fact that it is no longer resident.

Swapped pages do not form part of a snapshot and must be ignored during recovery. Consequently they are distinguished from other pages in the log by having *Swapped Page* meta records associated with them. However, swapped pages may later become part of a snapshot; if this occurs, a

*Snapshot Swapped Page* meta record containing the location of the swapped page is written to the log.

#### 4.3.6 Cleaning

Snapshotting causes new and modified data to be appended to the log and potentially makes some data already in the log redundant. Therefore, logs must be *cleaned* to recover space used by obsolete data. The log server cannot assist with this process since only store managers can interpret the data stored in segments. Therefore, store managers have the responsibility of cleaning the logical logs that they manage.

Another reason for cleaning is to re-cluster data in the store. This is advantageous for two reasons: firstly, if related data is grouped in segments, it makes reading more efficient. Secondly, if contiguous virtual pages are grouped into segments, it has the effect of compacting the B-Tree since a single node can describe an entire page range.

The process of cleaning, aims to compact *live data* into as few segments as possible whilst retaining appropriate clustering. In consultation with the APT and B-Tree, the store manager can examine arbitrary segments in the log and identify the live data. The store manager cleans by copying all live data from segments containing obsolete data into new segments. Once the new segments are secured on non-volatile storage, the cleaned segments may be returned to the log server using the *free_segment* operation. In our system individual managers may implement different cleaning policies.

Segment cleaning may be initiated in one of two ways: either by the manager reaching internal thresholds or by the manager being instructed to clean by some external agent. The manager may be instructed to commence cleaning if the log server is running low on clean segments, if the system is quiescent, by a timer or by a human manager.

The number of segments cleaned at a time depends on the manner in which cleaning has been triggered. For example, cleaning triggered by reaching a threshold is stopped when a second threshold is reached. The amount of cleaning performed when externally triggered is a parameter of the cleaning request.

## 5 Use of the Store in Grasshopper

The architecture of the persistent store described within this paper was designed for use in the Grasshopper persistent operating system. In Grasshopper all processes (termed *loci*) execute in the context of persistent data repositories known as *containers*. Loci are free to migrate between containers by *invoking* them. Thus a locus may mutate data in multiple containers. Each container has an associated *manager* which implements the functionality of the store managers described in Section 4.

Since loci can concurrently execute within containers, it is possible for them to share modified data; this introduces the possibility of a locus becoming causally dependent on others. Consequently, some mechanism must be provided to ensure that individual locus snapshots form a globally consistent state.

We have experimented with several different persistence regimes in Grasshopper. In our experimental system, we are currently exploring a locus based snapshot model. We are also experimenting with both eager and lazy approaches to global consistency: In the eager system the transitive closure of causally dependent loci are snapshotted together. In the lazy system, when a locus is snapshotted, its state and the state of any modified pages it has accessed must be written to the log. This process is mediated by the kernel which stores the kernel level data in its own log and instructs managers to snapshot the appropriate user level data from the containers they manage.

Globally consistent states are found by the kernel which informs managers of the existence of such states. Managers never perform B-Tree snapshots unless they are aware of a globally consistent state.

On restart, the kernel finds the latest consistent state from its own logical log and instructs the managers to restore the appropriate user level state. Finally, the loci found in the checkpointed kernel data may be restored to their last consistent state. In this manner the Grasshopper system supports resilient persistent processes.

## 6 Conclusions

We have examined the architecture of existing persistent stores and identified problems relating to scalability and efficiency. This has motivated the design of a new log-structured approach which is

both efficient and scalable. The approach is based on the use of a log server which provides clients with private, threaded logical logs.

The major advantage of our approach is the avoidance of large, persistent data structures to describe the state of the store. Our technique maintains minimal persistent meta data and uses transient data structures to describe the state of the store. This permits efficient snapshot processing and hence high throughput.

We have implemented the store described in this paper and it is used in the Grasshopper operating system to manage both user level and kernel level persistent data. In conjunction with our optimistic process checkpointing scheme [7], the store supports resilient persistent processes. Preliminary performance measurements confirm the efficacy of this approach.

## 7 References


[1] Atkinson, M. P., Bailey, P. J., Chisholm, K. J., Cockshott, W. P. and Morrison, R. "An Approach to Persistent Programming", *The Computer Journal*, Volume 26, Number 4, pages 360-365, 1983.

[2] Brown, A. L. "Persistent Object Stores", Ph.D thesis, Computational Science, University of St. Andrews, 1988.

[3] Challis, M. F. "Database Consistency and Integrity in a Multi-User Environment", *Databases: Improving Useability and Responsiveness*, Academic Press, pages 245-270, 1978.

[4] Davies, C. T. "Recovery Semantics for a DB/DC System", *ACM Annual Conference*, pages 136-141, 1973.

[5] Dearle, A., di Bona, R., Farrow, J., Henskens, F., Hulse, D., Lindström, A., Norris, S., Rosenberg, J. and Vaughan, F. "Protection in the Grasshopper Operating System", *Proceedings of the 6th International Workshop on Persistent Object Systems*, Tarascon, France, Springer-Verlag, pages 60-78, 1994.

[6] Dearle, A., di Bona, R., Farrow, J., Henskens, F., Lindström, A., Rosenberg, J. and Vaughan, F. "Grasshopper: An Orthogonally Persistent Operating System", *Computer Systems*, pages 289-312, Summer, 1994.

[7] Dearle, A. and Hulse, D. "On Page-based Optimistic Process Checkpointing", *IWOOOS '95*, Lund, Sweden, to appear, 1995.

[8] Elhardt, K. and Bayer, R. "A Database Cache for High Performance and Fast Restart in Database Systems", *Transactions on Database Systems*, Volume 9, Number 4, pages 503-525, 1984.

[9] Gray, J., McJones, P., Blasgen, M., Lindsay, B., Lorie, R., Price, T., Putzolu, F. and Traiger, I. "The Recovery Manager of the System R Database Manager", *Computing Surveys*, Volume 13, Number 2, pages 223-242, June 1981.

[10] Liedtke, J. "Address Space Sparsity and Fine Granularity,", *ACM Operating Systems Review*, Volume 29, Number 1, pages 87-90, 1995.

[11] Lindström, A. "Multiversioning and Logging in the Grasshopper Kernel Persistent Store", *IWOOOS '95*, Lund, Sweden, to appear, 1995.

[12] Lorie, R. A. "Physical Integrity in a Large Segmented Database", *ACM Transactions on Database Systems*, Volume 2, Number 1, pages 91-104, 1977.

[13] Munro, D. S. "On the Integration of Concurrency, Distribution and Persistence", Ph.D. thesis, Computational Science, University of St Andrews, 1993.

[14] Rosenblum, M. and Ousterhout, J. K. "The Design and Implementation of a Log-Structured File System", *13th ACM Symposium on Operating Systems Principles*, Pacific-Grove, California, *ACM Operating Systems Review*, Volume 25, Number 1, pages 1-15, 1991.

[15] Vaughan, F., Schunke, T., Koch, B., Dearle, A., Marlin, C. and Barter, C. "Casper: A Cached Architecture Supporting Persistence", *Computing Systems*, Volume 5, Number 3, California, pages 337-364, 1992.