A Flexible Persistent Architecture Permitting Trade-off Between Snapshot and Recovery Times

David Hulse

Department of Computing Science
University of Stirling
Stirling, Scotland
dave@cs.stir.ac.uk

Alan Dearle

Department of Computing Science
University of Stirling
Stirling, Scotland
al@cs.stir.ac.uk

Abstract

For persistent systems to be useful they must provide efficient snapshot and recovery mechanisms. This paper describes the design of the persistence architecture of the Grasshopper operating system which addresses these requirements. It shows how the major components of the architecture interact with one another and how it can be used to provide fast snapshot and recovery times. A novel feature of the architecture is the lazy recovery mechanism which delays the recovery of each component of the system until required.

Keywords: Persistence, snapshot, checkpointing, recoverable virtual memory, recoverable processes.

1. Introduction

For persistent systems to be useful they must provide *stable* and *resilient* storage for data. Stability is used to record the state of the system on non-volatile media and resilience permits the stable system to be restarted following a shutdown or crash. Persistent systems usually support stability by providing a *snapshot* or *checkpoint* mechanism and handle resilience with a *recovery* mechanism capable of reconstructing a consistent system state from a series of snapshots.

The efficiency of persistent systems can be measured by the speed of their snapshot and recovery mechanisms. Some applications, such as transaction processing systems, require high speed snapshotting to provide acceptable transactional throughput. Other applications are designed for high availability and benefit from fast recovery times. It is important that persistent systems be able to meet both these demands if the approach is to be used by these types of applications.

This paper describes the design of the persistence architecture of the Grasshopper operating system. It shows how the major components of the architecture interact with one another and how it can be used to provide fast snapshot and recovery times. The next section presents an overview of the facilities provided by the Grasshopper kernel. Section 3 discusses the design of the persistence architecture and describes the operation of its main components. Section 4 describes various strategies which can be employed to improve recovery time to adapt to the needs of the system being developed. Finally, Section 5 concludes the paper.

2. Grasshopper

Grasshopper is an operating system specifically designed to support the development of persistent systems [1]. It provides three basic abstractions, *containers*, *loci* and *capabilities*, all of which are inherently persistent.

Containers are the only storage abstraction provided by Grasshopper. They are coarse-grained persistent entities capable of holding very large quantities of data. They have properties similar to process address spaces in that data stored within them may be directly referenced using an address relative to the start of the container. However, unlike process address spaces, data stored within containers is persistent. In contrast to conventional systems in which address spaces are permanently associated with processes, containers are not associated with any form of process and may be completely passive data repositories.

In Grasshopper, the locus is the only abstraction over execution. Its name derives from the term *locus of execution* and was chosen to avoid ambiguity with existing concepts such processes and threads, neither of which truly capture the required semantics. Loci are scheduled by the kernel and execute within an address space that is composed of data stored within a particular container known as its *host*. Although each locus executes within a separate address space, many loci may share a common host container which allows them to share the stored data. A locus may move to a new host container via the *invocation* mechanism [1] which effectively changes its host container. For these reasons, containers and loci are completely orthogonal abstractions.

The persistence of data stored within containers is handled by *managers* which are responsible for their stability and resilience. Managers integrate these facilities with virtual address space management to provide an orthogonally persistent execution environment for loci. They are described more fully in Section 3.3.

Grasshopper uses capabilities as its only protection mechanism. Each capability refers to a particular locus or container and holds a set of rights which determine the operations that the holder of the capability can perform. Capabilities held by a container or locus are kept in a list maintained by the kernel. A locus may use any of the capabilities in its own list or any of those in the list of its host container. Since capabilities are kept within the kernel to prevent forgery, loci must refer to them by an index called a *Capref*. When performing system calls, loci must present the kernel with caprefs for any containers or loci that are affected by the operation.

3. Persistence Architecture

The purpose of this section is to describe the persistence architecture of Grasshopper. Figure 1 shows the main components of this architecture and the interactions between them. It depicts a locus executing within the context of a container which can hold both code and data. The persistence of this code and data is the responsibility of the container's manager which is actually a distinguished container housing the code and data related to the task of managing persistent data. As the system operates, the kernel requests the manager to handle events such as page faults, page evictions and snapshots of the data stored in the containers for which it is responsible. In return, the manager performs any system calls needed to complete its management tasks.

The Grasshopper kernel treats loci and the data accessed by them during computation as the unit of stabilisation. Loci are able to *snapshot* the state of their computation at any time, a task which is coordinated by the kernel and draws on services provided by the managers [2]. A snapshot consists of all the data related to the computation of a locus and includes:

1. any modified container data seen by the locus,

2. any data maintained within the kernel to represent the state of the locus and the containers in which it has executed.

Since a locus can move between containers during the course of its computation, a snapshot typically involves recording the state of pages within a number of different containers. In contrast to other persistent systems in which a snapshot involves making the entire persistent store stable, the snapshot mechanism in Grasshopper only affects the stability of the portions of containers seen during the computation of a particular locus.

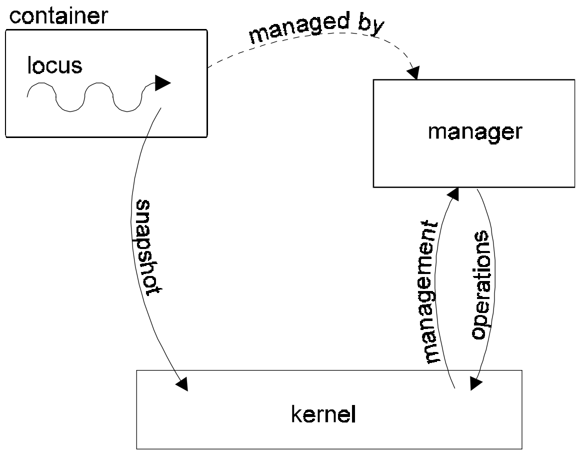

Figure 1. The components of Grasshopper's persistence architecture.

3.1 Stability Versus Recoverability

The computational model of Grasshopper allows many loci to share the same persistent data and operate upon it. Loci are free to use this shared memory environment as a means of inter-process communication. The effect of this is that the actions of one locus can be influenced by the actions of another locus. For example, one locus may write some information to the shared memory which is subsequently read and acted upon by another locus. In this way, the first locus has affected the path of computation taken by the second locus. This behaviour is often referred to as *causal* interaction. The significance of causal interaction is that it creates dependencies between loci. In the example above, the computation of the second locus becomes dependent on the computation of the first locus since if the first locus had taken a different course of action, and perhaps written different information to the shared memory, then the actions of the second locus may have differed also.

During the normal operation of the system it is possible to ignore the causal dependencies between loci because they are automatically preserved. However, if the system needs to be restarted after a shutdown or crash, locus snapshots must be used to rebuild the state of the system. If a locus is causally dependent on the computation of other loci at the time of making a snapshot, then that snapshot cannot be used at recovery time unless these other loci subsequently snapshot their computation since causal integrity would be violated.

To cope with this situation, persistent data in Grasshopper can exist in three different states. Data is in the *volatile* state when it is resident in main memory; performing a snapshot causes the data to become *stable* in that it is stored on non-volatile media. The snapshot remains in the *stable* state until all of its causal dependencies are satisfied by the existence of other stable snapshots containing the state of the necessary computation. Once this occurs, the snapshot is termed *recoverable* since it can now be used to recover the state of the system without violating causal integrity. The kernel is responsible for deciding when a snapshot becomes recoverable, but to do so it must keep track of causal dependencies between loci.

The kernel uses *vector time* [4] to represent the causal dependencies between loci. Each locus has an associated vector time which is lazily updated whenever a snapshot is performed [2]. The vector time contains a list of pairs representing the state of each computation on which the snapshotting locus is dependent. Each pair contains the identity of a locus and a timestamp derived from a Lamport clock [7] associated with the locus which is used to identify points in time during its execution. This information is sufficient to characterise the causal dependencies of loci and their snapshots [2].

3.2 Data Storage

The persistence architecture provided by Grasshopper is based on the use of a log-structured persistent store [5]. A module known as the *log server* maintains a log abstraction which may be read and written by multiple clients. Each client is given its own logical log and cannot examine or modify another client's log. In Grasshopper, the kernel and all of the managers are separate clients of the log server. Log entries are fixed size *segments* which contain a small amount of house-keeping information used by the log server but can otherwise be filled as needed.

In the architecture described in this paper, both the kernel and the managers fill segments with page sized blocks of data. In addition, *meta-data* is written into every segment by the client and acts as a description of the data in the segment. This meta-data is sufficient to permit any necessary data structures, such as address maps, to be rebuilt at recovery time if necessary. Therefore, a client's log serves as a self-describing, serial history of the state of one or more parts of the Grasshopper system.

3.3 The Internal Structure of Managers

Managers are the most important component of Grasshopper's persistence architecture. They are responsible for providing orthogonal persistence and integrating this concept with a virtual address space. Managers are distinguished containers which store the code and data needed to manage the persistence of container data. They provide an interface that the kernel and loci can call on to request management operations or general services. Each manager is capable of managing multiple containers simultaneously.

Since managers are implemented using containers, they to must be managed and as a result of this, some managers are capable of managing their own persistent data as well as that stored in other containers. These managers are known as *self-managing* and every Grasshopper system contains at least one such manager. Although these managers are largely responsible for themselves, it is not possible to be completely self-managing and it is necessary for the kernel to manage the persistence of a single page within the manager's container [8]. This page is termed the *stable page* and it contains the code and data needed to restart the manager after a shutdown or crash.

The kernel calls on managers to handle page faults occurring within the containers they manage. This requires them to allocate physical memory and perform the transfer of data to and from stable storage. Their services are also required when loci perform snapshots and when physical pages must be reused in the management of a virtual address space. The following sections describe the data structures used by managers to perform these tasks.

3.3.1 Container Management

One of the services a manager provides to loci is the ability to create containers. When a locus requires a new container, it requests the desired manager to create one. When a creation request is received, the manager must initialise its own data structures required to manage the container. When this is complete, it must register an identifier for the new container with the kernel; this is used when the kernel needs to request an operation on a particular container.

When creating a new container, the manager allocates an empty page within its own container. The address of this page is used as the registration identifier for the container. All of the information about the container can be reached from this page, and for this reason it is known as the container's *root page*. Since the manager is supplied with a container's registration identifier on every management request, it is trivial to locate the data structures representing the container to which an operation applies. The root page is used to store the following information, the layout of which is shown in Figure 2:

1. a pointer to the start of a modified page list, and
2. the location within the manager's log of a B-Tree representing the recoverable state of the container.

The *modified page list* is used to keep track of which container pages have been modified and which loci have read or written them. The list contains pairs each consisting of the identity of a locus and the address of a modified page within the container. An entry is added to the list every time a locus writes to a page or reads from a page already in the list. When the kernel requests the manager to snapshot the state of pages in the computation of a locus, this list is consulted to determine exactly which pages must be made stable.

The *B-Tree* is used to record the location in the log of each page in the recoverable state of the container. The manager uses the B-Tree to locate non-resident pages when servicing page faults. Each entry in the B-Tree represents a contiguous range of container pages and their location in the log. If it can be arranged that a contiguous range of container pages is mapped to a contiguous region in the log, then the B-Tree for the container can become quite concise. The manager is able to actively arrange pages in the log to prevent the B-Tree from growing too large.

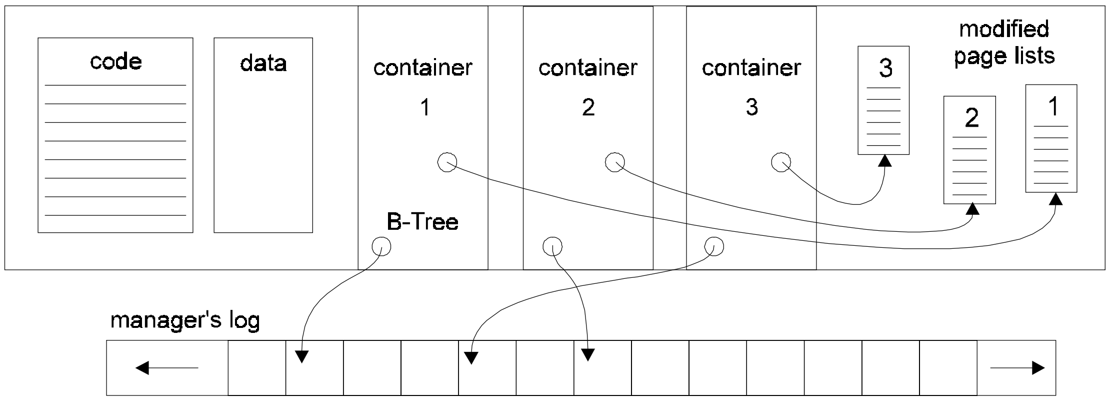

Figure 2. Data structures used to represent managed containers.

3.3.2 The Active Page Table

Having examined the data structures private to each managed container, it is possible to examine the global data structures used by managers. The most important of these is the *Active Page Table* or APT. This data structure resembles an inverted page table and is used to track the current location of both the volatile and stable versions of active pages within the containers being managed. The table is indexed by a key consisting of a container registration identifier and the address of a page.

Each entry in the APT contains the current state of a particular page and where it can be found. For example, resident pages are located within physical pages while stable pages are located within a particular segment in the log. The state of container pages is controlled by a finite state machine which is driven by system events such as page faults, page evictions, and snapshots. As each of these events occur, the APT is updated to reflect the changing states of the pages involved.

3.3.3 Address Resolution

The manager must be able to locate an arbitrary page within a container at any time. To do this it needs to know the registration identifier of the container and the address of the required page, both of which are passed as parameters. If the page is resident or has been resident in the past, then it can be located by consulting the APT. If it has never been resident then its location will be held in the container's B-Tree. To ensure location of the most up-to-date version of a page, the APT must be consulted before the container's B-Tree. The B-Tree is only used when there is no entry for the page in the APT.

3.4 The Internal Structure of the Kernel

The Grasshopper kernel is responsible for coordinating the stability and resilience mechanisms provided by the system. It contains persistent data representing the state of all the loci and containers in the system. This data is kept in a single persistent virtual address space within the kernel and is managed in much the same way as a manager would the address space of a container. Hence the kernel manager maintains an Active Page Table and B-Tree used to hold the location of pages within the recoverable state of the address space. Just as managers create a root page for each container they manage, the kernel has a root page for its persistent address space containing a pointer to the root of its B-Tree.

3.5 Snapshot Processing

Given an understanding of the internal structure of the kernel and the managers, it is possible to describe the snapshot mechanism. It is useful to remember that both loci and containers are persistent entities in Grasshopper. This means that the snapshot mechanism must capture the state of loci, containers and any data stored in those containers. As a result, when a locus performs a snapshot, it is snapshotting its computation which consists of any container data seen during execution along with the information stored in the kernel about its host containers and the locus itself. Since this information is handled partially by the kernel and partially by the managers, it is necessary for the kernel to coordinate the actions of each of these parties.

During the execution of a locus, the kernel and the managers monitor read and write faults to compile their *modified page lists*. As described earlier, these lists contain an entry for every modified page seen by a locus since its last snapshot. In addition, the kernel also maintains a list of containers in which a locus has seen modified data. The kernel uses this list to determine which managers it must interact with to process the snapshot request.

The first action taken by the kernel when processing a snapshot, is to seal off access to the pages comprising the state to be recorded. The kernel invalidates the pages from the contexts of existing readers and writers and then marks the pages with a flag indicating that they are undergoing snapshot processing. Any loci which fault on pages marked with this flag will be blocked until the state of the page has been recorded. This prevents any further pages seen by the snapshotting locus from being modified which is important because it stops the set of pages which must be part of the snapshot from growing and makes it possible to identify the set of loci on which the snapshotting locus is causally dependent. At this point, the kernel updates the vector time of the snapshotting locus to capture its causal dependencies. It then requests the appropriate managers to snapshot the modified pages stored within containers.

The kernel provides each manager with the identity of the snapshotting locus and the registration identifier of each container in which pages must be snapshotted. Managers use this information to determine from their modified page lists, exactly which pages must be snapshotted. The manager's task at this stage, is to record the state of each of these pages. Notice that there is no requirement that the pages be made stable at this time, only that a record of their state be taken. This might involve copying each of the pages into temporary buffers which are written to stable storage at a later time, or it might involve actually writing the pages to stable storage immediately. Another possibility is that of using *copy-on-write* to delay taking any action at all until the last possible moment. Briefly, the pages involved in the snapshot are marked read-only and it is only when a write fault is received on one of the pages that it is copied or written to stable storage. Results from other researchers [3] have shown that the use of copy-on-write techniques can improve the throughput of snapshots quite dramatically. The fact that pages need not be forced to stable storage immediately allows the managers to perform I/O asynchronously which presents the opportunity to make better use of the stable storage mechanism by clustering multiple snapshots into the same log segments. This helps to eliminate partially full segments and reduces the number of I/O requests that must be performed.

Once each manager has completed the snapshot request, the kernel removes the flag preventing access to the snapshotted pages thereby unblocking any loci waiting to access them. With the state of container data recorded, all that remains is to record the state of any pages managed by the kernel. These pages are processed using the same techniques as the managers and once complete, the flag preventing access is removed and the snapshot is effectively complete.

Next the kernel uses the vector time of the locus to decide whether the new snapshot is part of a recoverable state. This is determined by the algorithm of Johnson and Zwaenepoel [6] which maintains a *recovery vector* representing the most recent recoverable state. By examining the causal dependencies recorded in the new vector time, it either updates the recovery vector thereby incorporating the snapshot into the recoverable state, or determines which other loci must snapshot before the new snapshot can become recoverable. If the recovery vector is updated to reflect a new recoverable state, then it is possible that previous snapshots which depended on the new one have also become recoverable. Since managers are able to delay the writing of a snapshot, it is possible for a snapshot to become recoverable without actually being stable. To eliminate this possibility, the kernel performs further processing to guarantee that the recoverable state of the system actually moves forward in time by forcing the managers to make the snapshot fully stable.

Although the kernel is aware of the recoverable state of the system at any particular time during normal operation, this information is lost following a shutdown or system crash unless additional action is taken. If necessary, the snapshots written to the logs for the kernel and the managers contain enough information to enable the recoverable state to be reconstructed. This process can be time consuming and is described in the next section. Since it is desirable to avoid this situation, Section 4 describes techniques that can be used to provide faster recovery times.

3.6 System Recovery

When the Grasshopper kernel restarts, its job is to find the most recent recoverable system state and resume computation from that point. The first step is to rebuild the map containing the location of pages in the recoverable state of the kernel's persistent store which holds the information for all the containers and loci. The kernel must also determine which loci are recoverable. All of this information can be derived from the meta records in the kernel's log.

In the absence of any other help, the kernel must begin by scanning the meta information in each log segment starting with the first and working its way towards the end of the log. The meta information contains the virtual address of each page in the kernel's store along with its location in the log and the identity of the snapshot to which it belongs [5]. As each complete snapshot is found, entries are added to the kernel's Active Page Table (APT) to record the stable locations of the pages that comprise it.

It is also necessary to determine whether each snapshot is part of the recoverable system state by using the algorithm of Johnson and Zwaenepoel. The recovery vector is initially empty when the recovery process begins. As each snapshot is found, the kernel supplies the algorithm with the vector time of the locus that generated the snapshot. It is possible to locate the vector time using the snapshot identity which is actually the virtual address of the start of the kernel information for the locus that produced the snapshot. The vector time is stored at a known location within this information and is easily extracted. When a snapshot is found to be recoverable, the kernel updates its B-Tree which will eventually map the recoverable state of the kernel's persistent store. This is done by moving the APT entry for each page in the snapshot into the B-Tree; any entries remaining in the APT are for pages in snapshots which are not yet recoverable.

After processing all of the snapshots in the kernel's log, the B-Tree maps the recoverable state of the kernel's store and the recovery vector contains the identity of the loci which are to be resumed. At this point, the kernel has rebuilt the information it held before the shutdown or crash occurred and is now in a position to restore the system to a consistent state of operation. This entails restarting all of the managers and ensuring that the state of the data stored in containers is consistent with the recovered kernel state. However, since it is not known which containers and managers, will be required by the recovered loci, the kernel performs this step lazily by beginning to schedule the loci and using page faults to identify which containers need to be recovered.

During recovery the kernel passes the details of each page fault to the manager as usual. If the manager has not been restarted, its stable page will not be resident and another page fault will result. Since the kernel is responsible for the persistence of stable pages, it locates the page using the kernel information for the manager and makes it resident. The original page fault cannot be processed until the manager is given a chance to recover so the kernel postpones handling the fault and upcalls the manager notifying it of the need to restart.

When a manager receives restart notification, it uses information kept in its stable page to locate the pages containing its code and data segments and proceeds to prefetch them thereby avoiding several more unnecessary page faults. Like the kernel, a manager also requires a map specifying the location of the pages in the recoverable state of its address space. Furthermore, a similar map is required for each of the containers it is responsible for. In the worst case, these maps will need to be reconstructed from information stored in the manager's log; this situation is assumed in the description which follows.

To reconstruct the container maps, it is necessary to know which snapshots make up the recoverable states; this is determined by the identity of the recoverable loci. If a locus is recoverable, then any snapshot produced in its past is part of the recoverable state of some container. Therefore, the recoverable state of a container is composed of all the snapshots made by the recovered loci up until the time at which they are recovering. To assist with recovery, the kernel provides the manager with the identity of each recovering locus along with a list of the last snapshot identifier used for each container up until the recovery time of each locus. Since each snapshot in the manager's log is labelled with the identity of the generating locus and the snapshot identifiers are totally ordered, it is possible to scan the manager's log and pick out snapshots which are part of the recoverable state of each container. By examining the meta information in these snapshots, the manager can build a B-Tree representing the recoverable state of each container including its own.

Once a manager has rebuilt the container B-Trees, it can use the B-Tree for its own container to locate the root pages of the other containers for which it is responsible. This allows it to service management requests for these containers. At this point the manager is fully restarted and control can be returned to the kernel which is finally able to process the original page fault by passing its details on to the manager. As loci continue to run, their page faults eventually cause all of the required managers to restart consequently ensuring that the state of the managed containers is consistent with the recovered state of the kernel.

This section has described how the kernel is able to recover the system to a consistent state assuming that the logs contain nothing else apart from snapshot data and the associated meta information. Obviously, the longer the logs become, the longer the kernel and the managers must spend scanning them to rebuild their data structures describing the recoverable state. This situation is the worst case since it ignores the possibility of periodically writing extra information to the logs to avoid having to rebuild data structures such the B-Trees at recovery time. The next section describes this and other methods for reducing the system's recovery time.

4. Improving Recovery Time

The biggest impediment to fast recovery times is the need for the kernel and the managers to spend time scanning their logs to rebuild data structures describing the recoverable state of the system. The following sections describe techniques that can be used by the kernel and the managers to improve their respective recovery times.

4.1 Improving Kernel Recovery Time

To facilitate fast kernel recovery, it is necessary to avoid scanning as much of the log as possible. This can be accomplished by moving the point in the log at which scanning begins towards the end of the log. Before presenting a solution, it is necessary to consider what information the kernel requires in order to begin scanning its log at an arbitrary point.

In the previous section, it was assumed that the kernel began scanning at the start of its log, a point corresponding to the first bootstrap when their were no recoverable loci and the kernel's persistent store was empty. Together these facts comprise a specification of a recoverable state of the system, albeit a trivial one. From this specification or *baseline* it is possible to incrementally improve the recoverable state by examining snapshots found in the log. The important bits of information gained from this trivial baseline are the initial values of the recovery vector for Johnson and Zwaenepoel's algorithm and the kernel's B-Tree. Since the kernel maintains this information during the running of the system, it is possible to generate a baseline corresponding to a particular recoverable system state, simply by writing the information to the log.

Given the ability to generate baselines when desired, the only remaining problem is to arrange for the kernel to begin scanning the log from a point other than the beginning. Unfortunately, this point cannot necessarily coincide with the location of a baseline. When a baseline is written to the log, it is likely that there are some snapshots in the log which are not yet part of a recoverable state. These snapshots must be examined when scanning the log, otherwise it may be impossible for them to ever become part of a recoverable state. Therefore, the point at which scanning begins can only be moved forward as far as the earliest snapshot which is not yet recoverable. This point and the location of the most recent baseline are stored in the kernel’s root page which is examined as the first action upon restart.

Clearly the frequency with which baselines are generated affects the kernel’s recovery time. It is possible to implement policies which attempt to bound recovery time by generating baselines at appropriate intervals. Such policies may be based on measurements of log length or idle time or other such metrics.

4.2 Improving Manager Recovery Time

To fully recover after a shutdown or crash, a manager requires a B-Tree for each of the managed containers that reflects a state consistent with the recovered state of the kernel. Without additional information, these B-Trees must be rebuilt by examining the snapshots found in the log. It is possible to employ the same tactics used by the kernel to improve the recovery time of managers. However, the techniques used for managers are more complex since there is typically more than one B-Tree involved.

During the normal running of the system, a manager is able to update its own B-Tree and the B-Trees of the containers for which it is responsible from the entries held in its Active Page Table. This occurs in response to receiving notification from the kernel whenever a snapshot becomes part of the recoverable state. At any time, it is possible to write these B-Trees to the log, thereby creating baselines for the corresponding containers. The location of the most recently written B-Tree for each container is kept in the container’s root page. The location of the manager’s own B-Tree is kept in its stable page. At recovery time, the manager can locate its own B-Tree which holds the location of the root pages for each of the managed containers from which the other B-Trees can be located. If some of these B-Trees represent states earlier than the state to which the system is recovering, then the manager need only scan those snapshots which became recoverable after the B-Trees were written to the log.

The point at which the manager must begin scanning is stored within its stable page along with the location of its B-Tree. This point is determined by how recently the B-Trees for each container were written to the log. If no B-Trees have ever been written, then scanning must begin at the start of the log. When a B-Tree is written to the log, it effectively summarises a set of snapshots removing the need to ever scan them again. However, there are usually a set of snapshots which cannot be summarised by a B-Tree because they are not yet part of a recoverable state. The position of the earliest of these snapshots determines the location at which the manager must begin scanning upon restart. It is this location which is stored in the manager’s stable page.

5. Conclusion

This paper has presented the design of the persistence architecture used in the Grasshopper operating system. The use of managers to control much of the operation of the system provides flexibility since they can be tailor-written to suit the needs of individual applications. Additional flexibility is given by the snapshot and recovery mechanisms which provide a great deal of scope for implementing policy. This provides an ideal platform for investigating the effect of different snapshot and recovery policies.

The use of the lazy recovery mechanism driven by page faults is a novel part of the architecture. It allows a gradual recovery of the system rather than delaying restart by eagerly recovering parts of the system which might not be required immediately.

The implementation of the architecture described in this paper is nearing completion. Currently the system supports the worst case situation in which the kernel and the managers must rebuild their data structures by scanning their logs at restart. Further work will be aimed at the implementation of the techniques outlined in Section 4 and will entail an investigation of the best snapshot and recovery policies and the trade-offs involved.

References

[1] Dearle, A., di Bona, R., Farrow, J., Henskens, F., Lindström, A., Rosenberg, J. and Vaughan, F. “Grasshopper: An Orthogonally Persistent Operating System”, *Computer Systems*, vol Summer, 1994.

[2] Dearle, A. and Hulse, D. “On Page-based Optimistic Process Checkpointing”, in *Proceedings of the 4th International Workshop on Object-Orientation in Operating Systems*, Lund, Sweden, pp. 24-32, 1995.

[3] Elnozahy, E., Johnson, D. and Zwaenepoel, W. “The Performance of Consistent Checkpointing” in *Proceedings of the 11th Symposium on Reliable Distributed Systems*, Houston, Texas, IEEE, pp. 39-47, 1992.

[4] Fidge, C. “Timestamps in Message-Passing Systems That Preserve Partial Ordering” in *Proceedings of the 11th Australian Computer Science Conference*, University of Queensland, pp. 56-66, 1988.

[5] Hulse, D. and Dearle, A. “A Log-Structured Persistent Store” to appear in *Proceedings of the Australasian Computer Science Conference*, Melbourne, 1996.

[6] Johnson, D. and Zwaenepoel, W. “Recovery in Distributed Systems Using Optimistic Message Logging and Checkpointing”, *Journal of Algorithms*, vol 11, 3, pp. 462-491, 1990.

[7] Lamport, L. “Time, Clocks, and the Ordering of Events in a Distributed System”, *Communications of the ACM*, vol 21, 7, pp. 558-565, 1978.

[8] Lindström, A., Dearle, A., di Bona, R., Rosenberg, J. and Vaughan, F. “User-level Management of Persistent Data in the Grasshopper Operating System”, Technical Report GH-08, Department of Computing Science, University of Stirling, June 1994.