RT1R1/2: Report on the efficacy of Persistent Operating Systems in supporting Persistent Application Systems

David Hulse and Alan Dearle

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

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

1. Problems with Conventional Operating System support for Persistent Application Systems

Over the past fifteen years much research effort has been expended in attempting to build systems which provide orthogonal persistence. The idea behind persistence [5] is simple: all data in a system should be able to persist (survive) for as long as that data is required. Orthogonal persistence means that *all* data may be persistent and that data may be manipulated in a uniform manner regardless of the length of time it persists. In this sense, persistent systems provide a uniform abstraction over storage. Orthogonal persistence is not found in contemporary operating systems, nor in most programming languages or database systems. In these systems, long-lived data is treated in a fundamentally different manner from short-lived data. Traditionally, long term data is maintained in a database or file system and short-term data is managed by a programming language.

A number of experimental systems supporting orthogonal persistence have been constructed. These usually provide a large store within which concurrent processes may manipulate persistent data. In some of these systems, the stores contain all data including procedures, graphics objects, processes and their associated state. A common feature of persistent systems is that the persistent stores supporting them are both resilient and stable. Resiliency is usually achieved by making store updates non-destructive; this is generally implemented using logging [4] or some shadowing technique [31].

If persistent systems are to be anything other than research vehicles, they must be both stable and resilient. The persistent systems which exhibit these properties and that have been constructed to date, with a few exceptions, have been constructed on top of traditional operating systems. Existing operating systems do not provide an ideal platform for the development of persistent systems. This is not surprising since this was never part of their design goals. Indeed, most operating systems have files as their only abstraction over long term memory.

Tanenbaum [44] has listed the four major components of an operating system as being memory management, file system, input-output and process management. The nature of these four components is different in persistent systems. In a persistent system, the functionality of the file system and memory management are replaced by the persistent store. In many operating systems, input-output is presented using the same abstractions as the file system; clearly this is not appropriate in a persistent environment. Some persistent systems require that the state of a process persists; this is not easily supported using conventional operating systems. It is therefore to be expected that an operating system designed to support persistence will have a different structure from a conventional operating system and will provide a different set of facilities. We can summarise the principal requirements of such an operating system as follows:

1. The major requirement is support for persistent objects as the basic abstraction. Persistent objects consist of data and relationships with other persistent objects; the system must therefore provide a mechanism for supporting the creation and maintenance of these objects and relationships. This mechanism should be based upon a uniform addressing scheme used by all

processes to access objects. That is, all processes share a single logical address space. This is essential for orthogonal persistence.

2. A further requirement is that these objects must be both stable and resilient. The system must reliably manage the transition between long and short-term memory transparently to the programmer.
3. Processes must be integrated with the object space in such a way that process state is itself contained within persistent objects. The importance of this is that processes themselves become resilient.
4. Although the persistent store is uniform, there is still a requirement to be able to restrict access to objects for the same reasons that file systems contain access control mechanisms. Any operating system supporting persistence must therefore provide some protection mechanism.

We term an operating system that provides these facilities a *persistent operating system* [17]. In this section, we examine some existing persistent systems with respect to the four requirements listed above and discuss some of the approaches taken in supporting these facilities on top of conventional operating systems. This will provide a basis for the examination of the efficacy of persistent operating systems.

1.1 Support for persistent objects and relationships

Conventional operating systems expect to support processes that only access short term data in directly addressable physical or virtual memory. Long term data is held on backing store and cannot be directly addressed. In contrast, systems that provide orthogonal persistence treat all data identically as persistent objects. This leads to a requirement that objects must be addressed uniformly and moved between long and short term storage in a manner that is transparent to the application programmer.

Several approaches have been taken to solve this problem. In Brown's stable store [7] two address spaces are managed: a local process address space in which objects may be directly accessed by machine instructions and a software supported persistent address space; objects are transparently moved from one to the other on demand. This requires software address translation between local address spaces and the persistent address space; this software address translation can never be made as efficient as hardware address translation. However, the impact of the cost of address translation in persistent systems is not clear due to the lack of sufficient measurement. Indeed, Moss asserts that software address translation is more efficient than utilising hardware mechanisms under conventional operating systems [34].

The utilisation of paging mechanisms suffers from two problems. Firstly, no operating system constructed to date provides sufficiently flexible mechanisms to exploit the hardware facilities to their full potential. Mach [3] and Chorus [43], for example, do provide considerable flexibility in managing virtual memory but, as we will show later, do not deliver all the required functionality. Secondly, addresses supported by the conventional hardware may not be large enough for extremely large stores. We do not intend to directly address the latter problem in this paper, instead the reader is referred to Rosenberg et al [41].

1.1.1 Software address translation

The first object systems to be called persistent [6] [13] did not rely on any support from the hardware or operating system other than the provision of a file system. In this section, for illustration purposes, we will concentrate on one of these persistent object management systems, the CPOMS [8]. The CPOMS is the persistent object management system used to support the Unix PS-algol [1] implementations.

The persistent store implemented by the CPOMS is a large heap with objects being addressed using persistent identifiers known as PIDs. Normal pointers are traditionally referred to as local object numbers or LONs.

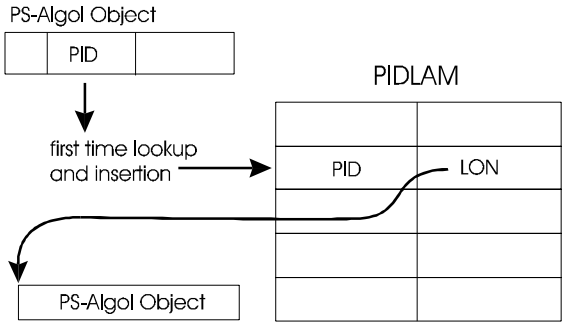

Figure 1**: Looking up a PID in the PIDLAM**

Since PIDs are pointers to objects outside of the program's address space, the objects to which they refer cannot be directly addressed by a PS-algol program. PIDs are identical in size to the normal pointers used by the PS-algol run time system but are distinguished by having their most significant bit set. Attempts to dereference PIDs are trapped and invoke the software translation mechanisms, whereby the object is fetched into memory and the PID is replaced by the appropriate LON; this activity has become known as *pointer swizzling*.

In order to prevent more than one copy being made of an object, a data structure called the PID to Local Address Map (PIDLAM) is kept. When a PID is first used and the object to which it refers is copied, the PID is entered into the PIDLAM along with the LON of the copy. Therefore, if the PID is used again, the LON of the copy can be found from the PIDLAM and used in its place as shown in Figures 1 and 2.

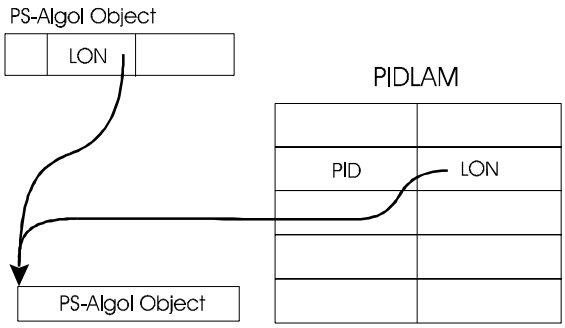

Figure 2**: Overwriting a PID by a LON**

The CPOMS address translation mechanism is relatively simple: a PID is divided into two parts: a segment identifier and an object number within that segment. Each PS-algol database is implemented using two Unix files: a data file and an index file. For each PS-algol database, the objects are stored in the data file which is composed of one or more segments. The index file maps database object numbers to addresses within the corresponding data file. Thus when a PID is encountered, the appropriate segment identifier is mapped to a particular database. Next, the database object number is calculated using the segment identifier and the object number within the PID. This number is used to address the database index file which yields the address of the desired object in the database data file.

Although relatively simple, this mechanism tends to be slow for two reasons: firstly all the address translation is performed in software, and secondly persistent object accesses involve several file seek and read operations. The second of these two problems may be addressed using memory-mapping techniques as discussed in the following section.

1.1.2 Memory mapping techniques

1.1.2.1 Using SunOS memory mapped files

Support for persistence in the Napier88 system [32] has, to date, been provided using several techniques [7]. The first uses a much-simplified CPOMS approach, the other uses SunOS memory mapped files. We will discuss this second implementation here.

The Stable Store which implements a resilient persistent address space as discussed in section 3.2.2 is maintained in a single, fixed length Unix file as shown in Figure 3. Using the SunOS system call *mmap*, the whole file is memory-mapped to a single virtual memory address range at an address *map_start*. The useable address space begins at *data_start* and extends to the end of the file. Reading a persistent address *x* constitutes accessing the contents of address *map_start + x* in the virtual memory of the executing process. The fetching of the appropriate page from disk is transparently handled by the operating system mapping mechanism.

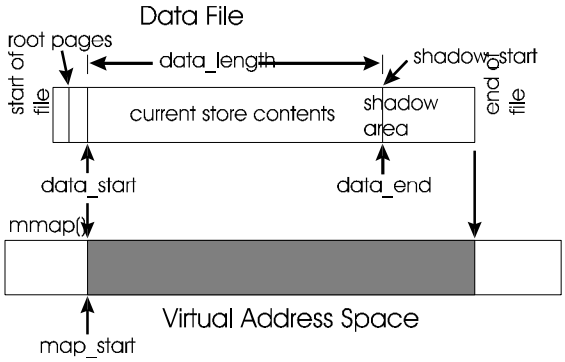

Figure 3**: The memory mapped Napier88 store**

This technique allows the store to be coded at a higher level by utilising the existing operating system support for virtual memory to abstract over disk access. This approach, however, gives less control over:

1. the time at which pages are actually written to disk,
2. the clustering and re-clustering of data on pages and in segments, and
3. the partitioning of the store for control of garbage collection.

1.1.2.2 Mach and Chorus

Mach [3] and Chorus [43] provide greater control over virtual memory in that both provide support for programmable page fault handling. In both systems, the user is permitted to provide a process which services page faults. This process is known as an *external pager* in Mach and as a *mapper* in Chorus. Since the Mach and Chorus communities use different terminology for what is essentially the same abstraction we will arbitrarily use the Mach terminology henceforth.

Both systems support an abstraction over memory called a *memory object*. Memory objects may be mapped into the virtual address space at any address. Both systems provide default mechanisms for the management of memory objects, for example a default pager which implements classical demand paging to a swap region. A user, however, may provide a manager for a memory object using an external pager. External pagers must conform to an interface specified by the kernel and are responsible for handling memory management requests from the kernel. For example, in response to an action by the swapper, the kernel may remove a page from physical memory and notify the external pager of its action. In this case the kernel will call the external pager interface function *memory_object_data_write* with suitable parameters. The external pager must deal with this page in some suitable way. The external pager interface provides enough functionality to allow page level access control over memory objects.

This mechanism may be exploited to implement stable stores. For example, at the University of Adelaide, the Mach external pager mechanism has been used to support a coherent distributed

persistent address space [25, 48]. In this system, a centralised store may be accessed by client processes distributed around a local area network in a manner similar to the systems described by Li [26] and Henskens [22].

However, a significant inconvenience is the kernel's removal of pages according to its own LRU algorithm. It would be more useful if the kernel requested the external pager to remove one or more pages, rather than sending its own choice of pages to the external pager for removal. This is due to the fact that the pages selected may contain pointers into the client's local heap area, in which case removal is a costly operation.

1.1.3 Conclusions

In this section we have attempted to show a progression of persistent address translation techniques ranging from purely software approaches through to the utilisation of hardware made possible using systems such as Mach and Chorus. The trend in operating system support has been to make more of the functionality provided by the hardware available to user applications. Indeed, much of the contribution of operating systems like Mach and Chorus could be considered to be finding suitable abstractions over the base functionality provided by the hardware.

The software approaches tend to be slow but permit the implementation of address spaces which are larger than the virtual address space size supported by the underlying hardware. The need for large address spaces has lead some designers to adopt hybrid approaches which use both pointer swizzling and hardware address translation, for example [36, 49].

The memory mapping abstraction provided by SunOS permits regions of the file space to be mapped into virtual memory and accessed transparently. However, this mechanism was not designed to support large address spaces such as those found in persistent systems. This leads to problems, for example the SunOS memory mapped file implementation of Napier88 suffers from the problem of slow start up times due to the operating system's eager construction of large virtual address maps.

We have indicated how Mach and Chorus both provide external pager mechanisms which permit the implementation of an application specific pager. Once again, this mechanism was not designed to support large persistent address spaces. Both these systems are deficient in that they do not permit total control over the virtual address space. In particular, in both systems it is the kernel that selects pages to remove from the process' address space when physical address space becomes short. This functionality causes many problems for the persistent system implementor who may be using memory in complex ways unknown to the kernel.

The systems described in this section demonstrate that it is possible to support a persistent system using standard paged address translation hardware. Mach and Chorus have shown that it is possible to provide useful abstractions over this hardware and this is seen as their main contribution to this field.

1.2 Stability and resilience mechanisms

The requirement for resilience is not peculiar to persistent systems; the same problem occurs with conventional file systems. For example, most operating systems provide limited recovery features such as *fsck* in Unix. However, the problem is perhaps more acute with persistent systems. In a conventional file system each file is essentially an independent object. Therefore, the loss of a single file following a crash does not threaten the integrity of the overall system. In a persistent system there may be arbitrary cross references between objects and thus the loss of a single object can result in total system failure. In this sense the problem of recovery within a persistent store is much more closely related to recovery in database systems [4].

Early persistent stores such as POMS [13] and the CPOMS [8] were constructed using conventional file systems with no special features. Since the underlying file systems offered no explicit support for persistence, and for stability in particular, techniques similar to those developed for database systems were used. The persistent store was implemented as a series of databases against which a program could apply transactions. As described above, such databases corresponded to individual files as provided by the operating system. A record of transactions was maintained in a transaction log, which allowed them to be either *committed*, thus changing the state of the database, or *rolled back*, thus returning the database to some previous stable state. To apply modifications to the

database a commit operation had to be invoked by the program. The persistent store was thus partitioned in an unnatural way to provide correspondence with files, and details of this partitioning remained visible to the user.

The advent of memory mapped files allowed the use of virtual memory technology in accessing the filestore. This facilitated the use of *shadow paging* [31] as a means of ensuring store integrity. Many of the later systems are based on this technique [7, 42, 45, 46]. It is therefore instructive to examine this approach in order to establish the requirements of a persistent operating system.

1.2.1 Shadow paging

Resiliency requires that the persistent store evolves from one consistent state to another atomically. That is, in the event of a system failure, all the changes are either recorded or the system recovers to the previous stable state. A number of techniques have been developed for achieving stability, particularly in the context of database management systems [8, 13, 21, 31, 42, 45, 46]. The techniques differ in their efficiency with regard to the particular application area. However, there are two basic requirements. They are:

- the ability to perform an atomic update operation, and
- the ability to identify the old data and new data prior to the stabilise operation.

In order to explain how atomic update may be implemented we will assume the following:

1. There is a mapping table from virtual persistent store addresses to physical disk addresses (such an address map is required in systems where the virtual address space is not mapped in 1-to-1 correspondence with the physical address space). All the data in the system can be found using this mapping table.
2. On system start up and after each stabilise operation a new copy of the mapping table and the data is made. Updates are made to these copies. That is, the old data is never overwritten. Such a system is unrealistic since the copy operation is too expensive but it will serve as a model for explaining atomic update. We examine some actual implementations and it will be seen that an efficient implementation is possible.

Prior to a stabilise operation there are two sets of mapping tables and data - the new updated one and the one representing the state of the system at the previous checkpoint. Challis' algorithm [10] uses two fixed blocks with known disk addresses that usually record the two previous stabilised states of the system. These are known as the root blocks. The root blocks contain information that allows the system to find the mapping table for a stabilised state. The root blocks record the two previous stabilised states.

Each root block also contains a version number that enables the system to determine which contains the most recent state. This version number is written twice as the first and last word of the block. The atomic update operation entails overwriting the root with the oldest version number, and a pointer to the new updated mapping table. The space occupied by the old stabilised state may now be reused.

Challis' algorithm depends upon two critical points for safety. Firstly an error in an atomic update can only occur if the root block is written incorrectly. It is expected that if a disk write operation fails during the atomic update it will inform the system which can then take appropriate action immediately. If, however, the failure is more serious, the technique depends upon the version numbers at the start and end of the root block being different in order to detect failure.

On system startup the root blocks are inspected. If the version numbers are consistent within the root blocks, the most up-to-date version of the system can be found. If not, only one root block may have different version numbers at any one time unless a catastrophic failure has occurred, which in any case would have other implications for the integrity of the data. Thus, subject to the above proviso, the correct stable data can be identified.

Assuming that an atomic update can be performed by this mechanism we return to the question of making efficient copies of the data. There are a number of different techniques that have been implemented and we now examine a selection of these.

Thatte [45] has proposed a *recoverable virtual memory* as the basis of a uniform memory abstraction for object-oriented databases. With Thatte's scheme, each page on the disk is in one of two forms, called *singleton* and *sibling*. Singleton form is used for pages unlikely to be modified and is represented by a single copy of the page on disk. In the sibling form two disk blocks are allocated to the page. When a page is written to disk a timestamp is written with the page, either in the page header or in the page table. The timestamps are derived from a reliable continuously running counter.

The essence of the scheme is that at any time a checkpoint operation may be initiated. This saves the current state of the system in a consistent manner. The time of the last checkpoint is recorded in a root page which is written in a secure manner. When a write fault on a singleton page occurs, if the timestamp of the disk copy is earlier than the last checkpoint then it must be converted to sibling form and the modified page is written to a new disk block. Otherwise the singleton page is not part of the previous checkpoint and may be freely modified. A write fault for a sibling page results in the most recent of the two disk blocks which is still part of the checkpoint state being retained and the other page overwritten. In addition, at any time a sibling may be converted to a singleton (in order to conserve disk space) provided the timestamps of both disk blocks are before the last checkpoint. In this case the older of the two blocks is discarded. On a checkpoint, a transient root is created in the store. This contains the current state of the processor registers so that, following a failure, they may be restored as at the last checkpoint.

The above rules guarantee that for any page there will always be a copy of that page on disk as at the last checkpoint (if it existed). These checkpoint pages are not overwritten until a new checkpoint has been established. This new checkpoint will become the current checkpoint only when the root page is updated. This may be performed in an atomic operation using Challis' algorithm as described above. Following a system crash the system is rolled back to the checkpointed state by restoring the processor registers from the saved transient root object.

Thatte does not describe the operation of the page tables in such a system. It is not clear where they reside (in the store or outside of the store) or how free space is managed. This is a key issue, since following a crash we must ensure that no disk pages are lost. The only mention of the page tables is the suggestion that the timestamps may well be held in the page tables themselves. This would seem to be quite expensive since the timestamp is 64 bits and thus the page tables would be very large. In addition, for sibling pages two such timestamps (as well as two disk addresses) are required, further increasing the size of the tables.

The major difficulties with Thatte's scheme seem to be the size of the page tables and the extra disk space required to maintain the sibling pages. For siblings where the timestamp of both pages is before the last checkpoint, the older of the two pages is wasted space. As Thatte points out it is possible to recover this space by converting the pages to singletons but there is some cost associated with such a task.

1.2.1.1 Brown's stable store

The current implementation of Brown's store divides the disk storage into two regions as shown in Figure 3. The first contains the current version of each page and the second, called the shadow area, contains a copy of the original version of each page modified since the last checkpoint. On the first modification of any page following a checkpoint, a copy of the page is made on a disk.

The system does not take advantage of virtual memory page protection to detect the first write to a page, instead a call to the store interface function *can-modify* precedes any attempt to modify an object in the persistent virtual address space. If this is the first such update to the corresponding page since the store was last in a stable state, a copy of the page (called a *before look*) is made in the shadow region before permission to modify is granted. In addition, a data structure called the *written bitmap* is altered to record the fact that a shadow copy of this page currently exists, preventing multiple shadows. When the new version of the page is eventually written back to the store, it is written back to its original location. In this way, the new state of the store is built by overwriting the contents of the old, but the old state can be completely restored from the shadow copies should a failure occur between checkpoints.

A checkpoint takes place by copying every modified page back to disk and then clearing the shadow region. Following a crash the last checkpoint state can be re-instated by using the pages held in the shadow region and the written bitmap to restore all modified pages to their original states.

It should be noted that a failure to correctly call *can-modify* on each write may result in incorrect operation of the checkpoint mechanism. In this sense Brown's scheme can only be used with trusted systems.

The major difference between Brown's scheme and the scheme implemented by Thatte is that Brown maintains a before look and uses this to recover the data following a crash; the other scheme creates an *after look* leaving the original data unmodified. The major advantage of a before look is that the original order of the data is preserved, thus maintaining any existing locality within the data and potentially improving sequential access and access to very large structures. The cost is the overhead of copying pages before they are modified and the maintenance of the disk-based written bitmap. Similar clustering can be achieved using an after look technique as suggested by Lorie in his original paper.

1.2.2 Adelaide's coherent persistent address space

The system built at the University of Adelaide uses the Mach external pager. Objects in the store are addressed using virtual memory addresses, meaning that address translation hardware may be exploited. The system uses an *after look* shadow paging scheme to provide store stability, with the first modification of a page being detected by the address translation hardware.

In this architecture, a number of clients execute against a shared stable store using a coherency protocol that guarantees data integrity. The frequency of checkpoints in any one client is reduced by maintaining a record of those clients which must be considered dependent upon one another due to the fact that they share modified pages. Only clients which are considered to be dependent on one another in this fashion need be stabilised together.

A central stable store server maintains information regarding the distribution and modification status of pages held by the clients; among this information is a record of which clients are dependent on each other. Interdependent clients are termed *associates* and a set of mutually dependent clients is called an *association*. Each association has a corresponding *page list*, which identifies those pages modified by members of the association since their previous stabilisation; this information is used to incrementally build the associations.

A mapping is maintained that maps the address of a virtual page to its location in stable memory (i.e., disk). This mapping table is called the *Logical to Physical map* (L-P map). As shown in Figure 4, each entry in the L-P map contains three fields: the physical page location of the stable version of the page, the location of the shadow copy of the page (if one exists) and a single bit selecting which entry holds the address of the stable page. Since the L-P map must be robust, it is stored within the persistent store which it manages.

The stabilisation protocol is as follows: the modified pages are first written from the persistent address space to their shadow location on disk. In the normal course of events, modified pages may also be delivered to the stable store by the coherency mechanism if there is insufficient space for them within a client's physical memory. These pages are also written to their shadow locations and are regarded as having been written back as part of this stabilisation. After all modified pages have been written to their shadow sites, the remainder of the stabilisation must be synchronised with any other stabilisations which have reached the same stage. The stabilising association's page list is then traversed (recall that this holds the list of all modified pages which are to be stabilised during the current stabilisation) and the selection bit is flipped for each of these pages to indicate that the shadow version is to be used as the current version once the stabilisation is complete. Next, the L-P map entries containing modified selection bits are written back to the appropriate shadow locations. Like the other systems described in this paper, in this system, the store is described by the contents of one of two header pages and Challis' algorithm is used to move from one stable state to another.

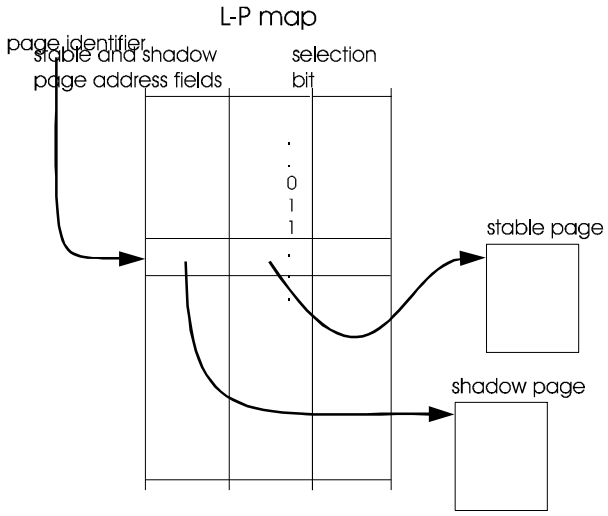

Figure 4**: The L-P map.**

It is possible for several independent stabilisation operations to be in progress at any time since, by definition, a client can only ever belong to one association. Consequently, pages from more than one stabilisation may be written to the stable store concurrently. However, care must be taken to ensure that the stable store moves from one stable state to another in an atomic fashion. In practice, the final stages of stabilisation must therefore be serialised. In particular, the L-P map can only make one flip at a time.

1.2.3 Conclusions

The checkpoint operations used to move persistent stores from one stable state to the next have been a problem for their implementors. Checkpointing the entire store at once as required by Thatte and Brown's schemes may have a detrimental effect on system performance because no store accesses may occur during a stabilise operation. However, the duration of stabilise operations may be minimised by ensuring that they occur frequently and that few modifications have occurred since the previous checkpoint. Such a strategy also has implications on system performance because a stabilise operation involves disk accesses. Ideally it should be possible to stabilise parts of the store separately, while still allowing other parts of the store to be used. This approach has been described above using lists of dependent clients called associations. Desirable operating system features to support the partitioned checkpointing of distributed server persistent stores are unclear, and are the subject of current research.

Like the problems encountered in address translation, the majority of problems in using conventional operating systems as platforms to support stability and resilience revolve around the abstractions provided by the operating system. We have shown that memory mapped files provide convenient mechanisms for accessing objects within files. However, if the system is to be made resilient, objects cannot destructively overwrite their original versions on disk. Therefore the conventional operating system mechanisms for flushing pages of a memory mapped file back to disk cannot be used. This results in much complexity being added to the code which supports resilience.

As we stated in section 2.3, a persistent operating system must provide a resilient persistent address space as a basic abstraction.

1.3 Process management and protection

Persistent systems provide a large persistent store in which all data resides. All processes execute against this store, meaning that it is necessary to ensure that a process may only access data for which it holds access permission. Failure to provide a protection mechanism could therefore result at best in the loss of privacy, and at worst in rogue or erroneous processes corrupting data owned by other users. Conventional systems such as Unix [37] implement two levels of protection of data; *file* level

protection using three tiered control over read/write/execute access, and *process* level protection using control of access to process address spaces.

File level protection is of little use in a persistent system where all data is represented as objects directly addressable by processes. Therefore, all protection must be provided at the process level. Indeed, it is common to find data encapsulated within processes, modules, abstract data types and other language constructs in persistent languages. However these schemes rely on the security of trusted system components such as compilers.

Originally, operating systems for hardware platforms supporting paged virtual memory, provided heavyweight processes, each running in its own address space and associated with a particular user. Such processes communicate with each other using various IPC mechanisms ranging from stream based systems (pipes and sockets) to shared memory and signals. The heavyweight process paradigm clearly does not meet our requirements which state that the persistent address space should be shared by all processes.

More recent operating systems, have added support for lightweight processes called threads, which operate within a single shared address space. These are more appropriate to our model but are all associated with the same user and they share the same protection privileges. All threads have access to the entire address space and it is therefore possible for one thread to accidentally (or deliberately) corrupt the data of another. When a thread requests a new segment to be mapped into its address space it immediately becomes accessible by all other threads sharing that address space.

A second problem with the process model supported by existing operating systems is that, unlike the file system, there is no notion of persistence. The process state is maintained in transient memory and in the event of system shutdown or failure this state is lost. In order to implement any form of persistence the programmer must provide explicit code to checkpoint the state of the process. This problem of the non-persistence of processes extends itself to login/logout time and results in many operating systems having a proliferation of startup files (e.g. login.com, .login, autoexec.bat, etc.) which effectively rebuild the process environment each time the user logs on to the system. It should also be noted that they only rebuild a statically defined environment; they do not recreate the dynamic environment of the user at the time of the last logout.

The dichotomy between temporary and permanent data found in conventional operating systems also manifests itself in the synchronisation primitives provided. Two distinct sets of synchronisation mechanisms are usually provided. The mechanisms supported for transient data are low-level, providing little support for transactions, roll-back, and stability. Conversely, the synchronisation mechanisms which operate on permanent data are more sophisticated but are intimately bound to the file system model. Despite this sophistication there is usually little support for atomic update, making the development of alternative data models above the file system difficult.

1.3.1 Conclusions

We have examined the process models available in operating systems constructed to date. There appear to be two different problems associated with processes. The first of these is support for a single shared address space; this is solved by the light weight thread model. The second problem is support for protection at the process level; this is a more difficult problem and there are four different solutions:

1. The use of type secure languages and trusted system components such as compilers which strictly enforce protection rules. This restricts the class of systems which may be supported, for example persistent C++ systems could not be safely used.

2. A second approach is to provide store level protection. For example, MONADS [39] provides protection based on *capabilities*. However, store-level protection requires some architectural support; this requirement is incompatible with our desire to implement a system on conventional hardware.

3. Another approach is to implement multiple persistent address spaces. Adopting this approach gives a coarse grain of protection – processes either have total access to the address space or none at all. This may be acceptable where multiple independent persistent systems are desired.

However, this approach is also incompatible with our requirements. It should be noted that with this approach it would be essential to have some global communication mechanism to allow processes operating in separate address spaces to cooperate. Such mechanisms have been described elsewhere [18].

4. A final approach is to associate a page protection list with each thread; in such a scheme, the page protection map is changed on a context switch. In this manner, the threads would share a single address space but may have their access restricted on a per page basis. This would result in performance penalties on machines with virtually addressed caches and would increase the overhead of a thread context switch. Although this mechanism does not provide access control at the object level, it does provide finer grain control over accesses than scheme *iii*.

It is apparent that none of these approaches fulfils the requirements stated in Section 1. In order to construct a persistent operating system on conventional hardware some compromises must be made. Approach *iii* permits the persistent operating system to support stores larger than the hardware supported address spaces. Approach *iv* has the advantage that it supports some level of protection within a persistent address space. A persistent operating system for stock hardware should support a combination of approaches *iii* and *iv*, permitting multiple address spaces with process based controlled access to pages.

1.4 The need for Persistent Operating Systems

The need for persistent operating systems has arisen because the construction of persistent application systems on top of conventional operating systems such as Unix, Mach, or Windows NT is prone to inefficiency [17]. This inefficiency is due to a *semantic gap* between the fundamental abstractions required to support orthogonal persistence and the abstractions provided by the underlying operating system. To overcome this mismatch of abstractions, it is common for the semantic gap to be bridged through the use of an abstract machine that maps the abstractions required by the persistent application system onto those provided by the operating system. Naturally, the introduction of an extra layer of software in this manner can only have an adverse effect on performance.

There are numerous examples of operating systems that support persistence in one form or another. Examples include *Multics* [2, 35] *MONADS* [24, 38, 39], *Eumul/L3* [27], *Clouds* [14], *Choices* [9], *KeyKOS* [20], and *Grasshopper* [40].

Each of these persistent operating systems attempts to address some or all of the issues described above.

2. Existing Persistent Operating Systems

This Section investigates several different operating systems that provide support for persistence. In each case, the abstractions relevant to the implementation of persistent systems are described. These are compared to the requirements of a persistent operating system as listed in the previous section.

2.1 Multics

In the early 1960's, hardware platforms were extremely limited in main memory capacity compared to the systems of today and most information was held on secondary storage. When data was needed for computation it would be formed into a *core image* and loaded into main memory for execution, much like a linked executable is today. If procedures and data were to be shared amongst users, separate copies were made in each core image. Thus, at any one time, main memory potentially contained many copies of the same information in separate computations, thereby making inefficient use of an already limited resource. These problems provided motivation for the *Multics* project, which was initiated at the Massachusetts Institute of Technology in 1964 [35]. Within a year, the project had attracted interest from a number of parties, including General Electric Corporation and Bell Laboratories.

The design of Multics is based around a virtual memory environment known as the *Multics Virtual Memory* [2]. The concept of the environment was based on the observation that significant gains in efficiency and reductions in program complexity could be realised if all stored information

were directly addressable by a processor. Efficiency would be gained by eliminating the duplication of procedures and data in core images by permitting computations to share a single resident copy. In addition, eliminating the need to transfer information between primary and secondary storage would reduce program complexity.

A Multics program operating in the virtual memory environment is able to directly address any information it requires subject to protection constraints. The Multics kernel transparently manages the movement of data between primary and secondary storage so that the only concept a program must deal with is the virtual memory. The structure and operation of this environment is now described in more detail.

2.1.1 The Multics Virtual Memory

The Multics Virtual Memory, pictured in Figure 5, is conceptually organised as a large collection of independent linear main memories known as *segments*. Segments may be thought of as a extension of *files*, being similar in that they are the unit of sharing, are referred to by a symbolic name, have a set of access attributes, and may grow and shrink as required. The single important difference between a file and a segment is that a segment is directly addressable by a processor whereas a file is not.

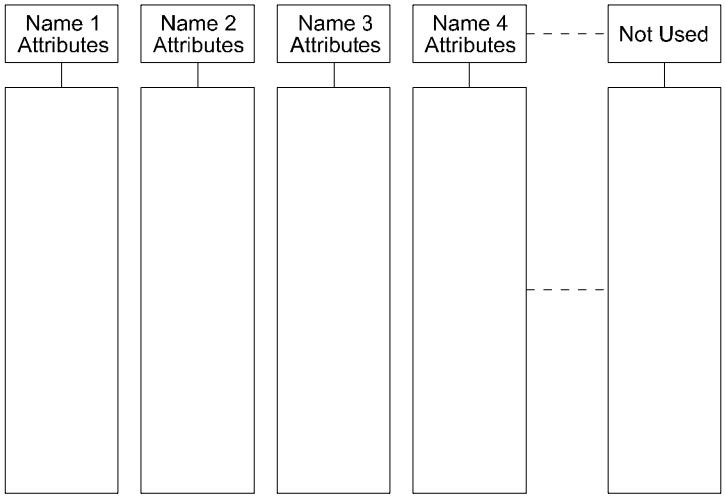

Figure 5: **Conceptual view of the Multics Virtual Memory.**

Each segment has a set of access rights that control the manner in which it may be used. Like files, these rights permit reading, writing, execution, and appending. Information within a segment may be referenced using a pair (*segment*, *offset*) in which *segment* is a descriptor for the segment and *offset* is the location of the required information relative to the start of the segment. Each such reference is checked by the hardware to ensure that the user has sufficient access privileges. The implementation of the virtual memory environment exploited the segmentation and paging features of the GE 645 processor.

2.1.2 Memory Management Features of the GE 645 Processor

The GE 645 is a segmented architecture in which each segment provides a separate paged virtual address space. All memory references are of the form [*s*, *i*] where *s* is the segment number and *i* is the index within the segment.

The word stored at [*s*, *i*] is located using several tables and registers defined by the hardware. Ignoring the paging mechanism, *s* is used as an index into a table called a *descriptor segment* (DS) which is held in main memory at a location pointed to by a special register called the *descriptor base register* (DBR). The DBR also contains the size of the descriptor segment to which it points so that segment numbers that are out of range can be detected. Each entry in the DS is known as a *segment descriptor word* (SDW) and contains a flag indicating whether the segment is “missing” or resident, the access rights, the length of the segment, and the core address of the first word of the segment. The

SDW at entry *s* in the DS is checked to ensure that the segment is not "missing", that the type of access required is compatible with the stored access rights, and that the index *i* is within the bounds of the segment as defined by its length. If so, the required word may be accessed at the address computed from the base of the segment stored in the SDW and the index *i*; otherwise an access fault is generated.

This description of segmentation ignores the operation of the paging mechanism for simplicity. In reality, each segment represents a paged virtual address space that is described by a *page table* located using the core address field of the SDW. Access to word *i* within a segment is performed by looking up the entry for the page containing *i* in the segment's page table. This entry contains a flag indicating whether the page is currently resident and if so, the core address of the page. If the required page is not resident, a fault is generated; otherwise the word can be accessed using the physical address of the page and the offset within the page. The tables and registers used to access the word stored at [*s*, *i*] are shown in Figure 6.

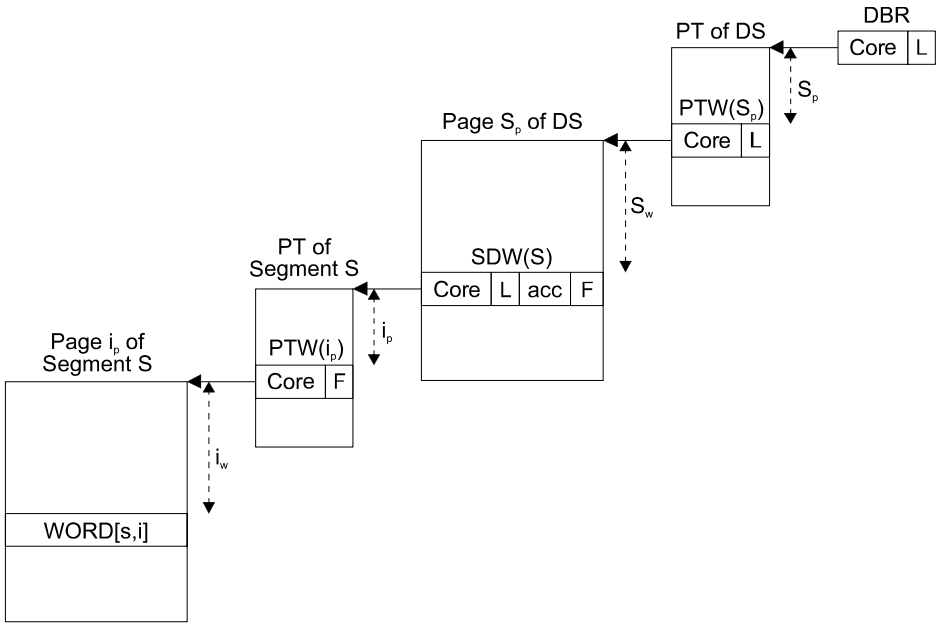

Figure 6: The operation of the se**g**mentation and paging features of the GE 645.

2.1.3 Computation in Multics

The Multics kernel provides the concept of a *process* with which computation may be performed. The segmentation and paging features of the GE 645 are used to provide each process with its own address space such that a one-to-one correspondence between processes and address spaces exists. The address space of a process is defined by a descriptor segment containing entries for the segments it is permitted to access. For security reasons, this descriptor segment is managed by the Multics kernel and is not addressable by the process.

Processes manipulate segments using their symbolic names. However, the processor can only refer to a segment via its index in the descriptor segment. Therefore, Multics provides a data structure called the *Known Segment Table* (KST) which maps between symbolic names and segment numbers for each process. If an attempt is made to use a segment whose symbolic name is not found in the KST, the Multics kernel attempts to make the segment *known* to the process by creating an entry in the KST.

The task of making a segment known to a process introduces the Multics *file system*, which is used to locate segments based on their symbolic names. Conceptually, the file system is a table containing one entry for each segment in the system. Each entry, called a *branch*, contains:

- The symbolic name of a segment.
- A unique identifier for the segment.

- The length of the segment.
- A list of users and their associated access rights.
- A map used to locate the pages of the segment on secondary-storage.

Branches are the primary source of information about segments. Thus, making a segment *known* to a process involves locating its branch within the file system using the symbolic name as an index. Once the branch is found, entries are made in the KST and DS of the process. The DS entry is initialised such that a *missing segment* fault will occur when the segment is first accessed. Since handling this fault requires information in the segment's branch, a pointer to the branch is kept within the KST entry. Making a segment known to a process merely allows the process to reference it using its symbolic name. To access a known segment, the segment must be resident (*activated*) and *connected* to the process. These concepts require explanation of a further data structure known as the *Active Segment Table* (AST).

The AST is a global data structure used to ensure that all processes share a single copy of each segment. Each entry in the AST describes a resident segment and contains a page table, a pointer to the segment's branch, a copy of the segment map from the branch, and the current length of the segment. *Activating* a segment simply requires the construction of an entry in the AST. However, since the number of AST entries is limited, activating one segment may involve deactivating another.

Once a segment has been activated, processes can *connect* to it through its entry in the AST. Connection involves filling in the segment's descriptor word in the DS of the connecting process. This requires information from the segment's AST entry, which is found using two previously unmentioned fields in the segment's branch. These are a flag specifying whether the segment is currently active, and if so, a pointer to the AST entry. The flag is used to determine whether a segment requires activation and the pointer field enables the AST entry of an activated segment to be found. Using the AST entry, the appropriate SDW can be properly initialised with a pointer to the segment's page table. To support the deactivation of a segment, a pointer to the newly connected SDW is entered into a list in the AST entry. The relationship between the various data structures used in the implementation of the Multics Virtual Memory is shown in Figure 7. The diagram illustrates the tables and entries required to connect a single segment to a particular process.

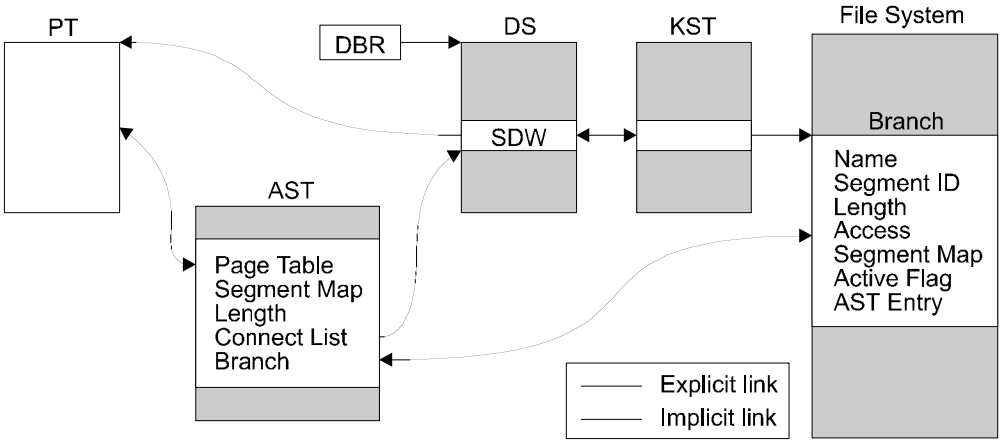

Figure 7: **Data structures used to manage the Multics Virtual Memory.**

2.1.4 Analysis

A principle objective of Multics was to allow a processor to directly address all information stored online. To achieve this, the operating system transparently manages the movement of data between primary and secondary storage on demand. Since the virtual memory is backed by secondary storage, stored data may live for arbitrarily long times. The result of this is that programmers need

only deal with a single storage abstraction, namely the virtual memory. Furthermore, data stored in the virtual memory may be manipulated in a uniform manner regardless of its intended lifetime. Therefore, the Multics Virtual Memory supports orthogonal persistence as a basic abstraction.

The virtual memory is organised into *segments* that can range in size from 0 to 64 KB. Segments are primarily used to control access to the virtual memory. Each segment has an associated *Access Control List* (ACL) containing entries that describe the rights granted to each user. An entry in the ACL consists of a user identifier and a set of access flags granting the ability to read, write, execute, or append to the segment. The Multics kernel exploits the hardware of the GE 645 processor to enforce the protection on a segment.

The data stored in a segment is paged on demand from secondary storage by the kernel. When dirty pages are expelled from main memory, they are written back to the location on secondary storage from which they were initially read, thereby overwriting the previous version. A problem with this technique is that the representation of a segment on secondary storage is not consistent unless all dirty pages have been flushed from main memory. Therefore, in the event of a failure resulting in the loss of main memory, the contents of secondary storage may be left in an inconsistent state; this seems to be dealt with via regular backups.

Multics keeps track of all active processes in a global data structure called the *Active Process Table* (APT). It is unclear whether the APT is part of the virtual memory (i.e. stored in a segment) and hence persistent. In principle, the APT could be stored in a segment and result in persistent processes.

To summarise, Multics provides orthogonally persistent storage in the form of segments and allows access to these segments to be controlled on a per-user basis. The paging mechanism used to automate the movement of persistent data between primary and secondary storage provides for the stability of data but does not guarantee resilience in the event of an unscheduled shutdown (crash). Processes are probably not persistent although this is possible in principle.

2.2 Monads

The MONADS project [24, 38, 39] began at Monash University in 1976 and subsequently spread to the Universities of Newcastle (Australia), Sydney, and Ulm. The aim of the project was to provide an environment for software engineering using data encapsulation and information hiding. These features were supported by a custom built hardware architecture, the main features of which were a capability-based addressing scheme, large virtual addresses, and a procedure-calling mechanism. These features are described in the following Sections.

2.2.1 Capability-Based Addressing

The idea of capability-based addressing is to assign every object in the system a unique address called a *Capability*. Capabilities cannot be forged and are never reused even if the objects to which they refer are deleted. The operating system and hardware protect the use of capabilities such that they cannot be modified or directly generated by programs. They may only be created and transferred under the control of the operating system. A program may only access an object if it holds a capability for the object.

2.2.2 Virtual Addressing Environment

The MONADS architecture actually comprises two systems named MONADS-PC [39] and [41]. The first of these machines supports 60-bit virtual address whilst the second supports 128-bit virtual addresses. The reason for these long addresses was to support the design philosophy of a uniform virtual memory in which all of the connected secondary storage could be referenced.

The virtual addresses are divided into two parts, an *address space number* and an *offset*. The address space number specifies a particular address space that is used to hold related data such as the code and data of a module or a stack. When an address space is created it is assigned a unique number that is never reused. Address spaces are paged to and from secondary storage; the MONADS-PC uses a 4 KB page size whilst the MONADS-MM provides both 4 KB and 64 MB page sizes. The offset portion of the virtual address specifies a particular byte relative to the start of the address space.

Address spaces may be divided into a number of *segments*, which are orthogonal to the paging mechanism. Segments are defined by capabilities held within *segment lists*. Each entry in a segment list specifies an address space number, a base and limit defining the extent of the segment within the address space, and a set of access rights. The capabilities in a segment list define the visible regions within an address space.

The addressing environment of a process is defined by a table containing entries known as *bases.* Each such entry points at a segment list thereby granting the process access to the segments defined within it. Therefore, at any point in time, a process may address any of the segments defined by the segment lists in its current table of bases.

2.2.3 Procedure Calling Mechanism

The MONADS procedure calling mechanism is used to support communication between information hiding modules. The code and data within each module is defined by a set of bases. When a process is executing within a particular module, the bases for that module are used to create the addressing environment of the process thereby ensuring that only the code and data within the module are visible. When the process performs a procedure call to another module, the addressing environment must be changed to that of the new module. This is achieved by switching the current set of bases to a new set defined by the module being called. On return from a procedure call, the original set of bases is restored thereby recreating the addressing environment of the calling module.

2.2.4 Persistence

MONADS supports the notion of persistence via the segment abstraction which provides support for large-grained persistent objects. In this regard, the virtual memory environment of MONADS is much like that of Multics. Segments are implemented by the innermost core of the MONADS operating system which runs on top of the bare hardware. The persistence mechanisms are integrated with the operation of the virtual memory such that the contents of secondary storage may be used for recovery purposes. The upper layers of the operating system, which implement the process and module abstractions, use segments to store internal data structures related to their implementation. Hence, processes and modules are both persistent.

To support recovery, the MONADS system periodically performs a global checkpoint in which all modified data is flushed to secondary storage. This approach was not based on any well-defined recovery model and led to significant delays during the normal operation of the system. This latter problem was recently addressed by the work of Jalili and Henskens [23] and is described in detail in Section 9.4.1.4. Briefly however, their work introduces a process-oriented approach in which individual processes can be checkpointed in a manner ensuring overall consistency. When one process checkpoints, all of the processes and data on which it depends are also checkpointed. If dependencies are few, only a small proportion of the system need be forced to secondary storage, which will clearly result in improved checkpointing performance.

2.2.5 Analysis

The MONADS system is based on the *segment* abstraction which provides a data storage mechanism supporting orthogonal persistence. Segments are used by the operating system for storage of important meta-data such as the state of processes. This means that all forms of data and computation are persistent within MONADS. In addition, the custom hardware on which the entire system is built contains support for capabilities, which are used to control access to segments and modules. These features satisfy three of the four requirements of a persistent operating system. The fourth relates to the stability and resilience of the persistent objects (segments in this case). The stability of segments is provided through the virtual memory paging mechanisms in conjunction with a checkpoint mechanism that could be used to produce a consistent state on secondary storage. In the absence of failure, this enabled the system to be restarted using the consistent state produced at the last checkpoint. However, failures could easily leave secondary storage in an inconsistent state such that recovery would no longer be possible. Thus, like Multics, resilience was a problem in MONADS.

The main problem with MONADS is the custom hardware platform on which it is based. The hardware design continues to age and hence suffers from poor performance and reliability compared to the modern platforms of today. This limits its usefulness and makes it difficult for others to benefit from the results of the research.

2.3 Clouds

The Clouds system [14, 15] was developed at the Georgia Institute of Technology in the mid 1980's as a research project aimed at exploring the realms of distributed computing. Clouds is a distributed operating system designed to run without other support on bare hardware. The first version of Clouds ran on VAX hardware and consisted of a monolithic kernel written in C. This version was subsequently abandoned due to its complexity and replaced with an improved design which benefited from the experience gained during the first attempt. The second version of Clouds is based on a minimal micro-kernel called *Ra* that runs on Sun 3/60 workstations and is written in C++. The basic abstractions provided by Clouds are *objects*, which abstract over storage, and *threads*, which abstract over computation. All data, programs, devices, and resources are encapsulated in objects, which are purely passive entities. Activity is provided by threads, which execute within objects.

A Clouds object is a persistent virtual address space which is not tied to any form of computation and is best suited for the storage of large-grained data and programs. The contents of an object are protected such that stored data may only be accessed by code within the object. Each object may provide a procedural interface via a set of entry points at which threads may commence execution. Code accessible through such an entry point is known as an *operation*. Each object has a global system-level name which is a unique bit-string that provides a location independent form of naming. User-level symbolic names are mapped onto system names using a nameserver. Each Clouds object is implemented as a single-level store that is demand-paged from a backing store. In this respect, objects are similar to the concept of segments in Multics and MONADS. The backing store for objects is provided by an abstraction called a *partition*, which is responsible for managing the paging of segments to and from secondary storage. A network partition is used which communicates with a Unix file server and stores data in files on a remote machine.

Threads are the only form of computation in Clouds and, unlike objects, are not persistent. They execute within the context of an object and may manipulate the data within it. However, threads may also move between objects by *invoking* an entry point of another object. The invocation mechanism is much like a procedure call except that the addressing environment of the thread is switched on call and return such that it may only access code and data within the current object. During an invocation, a thread may pass parameters in and out of an object thereby enabling the transfer of data between objects. Since Clouds is a distributed operating system, objects may reside on machines anywhere in the network and an invoking thread may move transparently across machine boundaries during invocations.

The model provided by objects and threads in conjunction with the invocation mechanism is sometimes referred to as the *object/thread* paradigm. It is an alternative to the *message passing* paradigm which is extremely common in distributed systems. The use of the object/thread paradigm completely eliminates the need for messages because communication between objects occurs via the invocation mechanism. The object/thread paradigm supported by Clouds permits multiple threads to execute concurrently within a particular object. To ensure the integrity of the system, the invocation mechanism is augmented with a consistency control system [11] that provides semantics similar to nested transactions [33]. It operates by labelling each thread and entry point with a particular "flavour" of consistency. The supported types are *Global Consistency Preserving* (GCP), *Local Consistency Preserving* (LCP), and *Standard* (S). In simple terms, when a thread invokes a particular entry point, the thread's label changes to that of the entry point and on return, it reverts to its previous value. When a thread's label transforms in this manner, it has implications for consistency control which depend on the label of the entry point.

LCP entry points commit updates each time a thread returns from an invocation. Therefore, LCP invocations are guaranteed to preserve self-consistency of the object. In certain situations, one object may contain code that invokes entry points of other objects and which wishes to maintain the consistency of the set of objects as a whole rather than each object individually. This is the purpose of

the GCP label. A thread returning from a GCP invocation does not commit its changes unless the entry point to which it is returning is labelled other than GCP. When this occurs, the thread's label reverts to that of the calling entry point and the changes made during all of the GCP invocations are committed together. Thus the set of objects involved is guaranteed to be mutually consistent as well as individually self-consistent. The detection and locking of modified data within objects is performed automatically by the Clouds kernel using faults generated by the virtual memory system. Locks are released automatically on return from invocation when the data is finally committed.

The remaining "flavour" of consistency is provided by the Standard (S) label. All threads are initially labelled S, which means that no consistency is guaranteed. The purpose of the S label is to permit low overhead computation in which consistency is not important. When an S thread invokes an LCP or GCP operation, its label transforms appropriately and any modified data is automatically locked and committed as required.

The interesting aspect of this consistency mechanism is the interaction between S threads and GCP/LCP threads. S threads have the property that they always access the latest version of data, which might contain uncommitted changes made by LCP and GCP threads. They are able to read and write such data without obtaining the appropriate locks as their LCP and GCP counterparts must do. This is unfortunate since S threads can potentially threaten the integrity of the system because they may violate the consistency semantics required by the LCP and GCP operations. To prevent this from occurring, a fourth type of consistency label known as *Inherited* (I) may be applied to entry points. When a thread invokes such an entry point, the invocation inherits the consistency label of the thread. Thus, an entry point labelled (I) invoked by a GCP thread will act as if it were labelled GCP. To provide further flexibility, an entry point labelled (I) may check the consistency label of an invoking thread and perform different actions accordingly. This makes it possible to filter out invocations from threads whose label does not meet certain criteria. In general, the interaction between S threads and GCP/LCP threads is well defined and mechanisms exist to enforce the integrity of the system in the case of this type of interaction.

2.3.1 Analysis

Clouds meets a number of the requirements of a persistent operating system, the first of which is the provision of persistent objects as a basic abstraction. Stability of objects is provided via the virtual memory system in conjunction with the partition abstraction. This stability mechanism is similar to that used by Multics and MONADS so it is likely that Clouds will also experience the same difficulties with resilience of objects as those systems. The persistence model in Clouds does not extend to threads, which must be restarted after failure. Finally, protection is provided by a capability mechanism, which controls access to various aspects of the system such as the invocation mechanism.

2.4 Eumel/L3

Eumel, and its successor L3 are part of a long running project begun in 1979 at the GMD in Germany. This project is the only operating system for conventional hardware other then Grasshopper in which support for orthogonal persistence is an explicit aim. As stated in []:

> "*We did not find any conceptual reason for anything not to be persistent. Obviously all things should live for as long as they are needed; and equally obviously turning off the power should neither be connected with the lifetime of a file nor with the lifetime of a local variable on a program stack.*"

The programming model provided by Eumel/L3 is similar to many modern micro-kernels. A computation or task, is comprised of a virtual address space and at least one thread that executes within that address space. Date is kept in data spaces that must be mapped into an address space to be accessed.

Communication is achieved by message passing between tasks. Rather than have explicit channels such as sockets in Unix or ports in Mach, a message is sent directly to a task by specifying its globally unique identifier. Also in contrast to most other systems, message passing in Emmel/L3 is synchronous; the sending thread waits in the kernel until a message is received. The chief advantage of

this approach is that no message buffering is required. As a result of this simple design the communication mechanism is very fast.

Persistence in Eumel/L3 is achieved by periodically taking a snapshot, or fixpoint, of the entire state of the machine. This includes the state of all data spaces and threads. After a crash, programmers need never perform any data recovery nor restart computations. The system restarts from the most recent fixpoint.

2.4.1 Analysis

To alleviate the overhead associated with taking a fixpoint, a concurrent checkpoint scheme similar to [19] is used. The implementers claim that there is virtually no interruption to on-going computations but only if about 30% extra physical memory is provided. This memory is used to accommodate copies of pages that are modified before they are written to disk. Of course, being a persistent operating system, Eumel/L3 has no file system, eliminating the need for file buffers and so the extent of this problem may be overstated. Nevertheless, performance might be improved by employing a finer grain checkpointing mechanism. A process oriented scheme such as those used in CASPER and Monads would reduce the need for extra memory because the need to perform global checkpoints is eliminated.

Unfortunately, it is not clear how this could be achieved in Eumel/L3 because it is not possible to isolate the effects of one computation from another. For example, consider three tasks T1, T2, and DB. T1 and T2 are clients of DB, which maintains a database. To update the database, T1 and T2 send messages to DB. For example, T1 changes the value of X to 4 while T2 changes the value of Y to 5. Both these changes are effected by a single thread in DB. To form a consistent checkpoint of T1, it is necessary to save its state plus the state of X because T1 is causally dependent on X. For the opposite reason, it is not necessary to save the state of Y. In Eumel/L3 this is impossible to detect because there is no notion of computational identity between address spaces. Sine the update of the database is performed by a thread in DB, there is no way of attributing the updates of individual items to the threads originating the requests. Consequently, Eumel/L3 employs a global checkpointing regime.

In a procedure-oriented system such as Monads, Clouds or Grasshopper computations move between address spaces and thus it is possible to keep track of which data they update. When a computation is checkpointed, it is possible to limit the amount of data written by finding a consistent cut with respect to just that computation.

2.5 Grasshopper

Grasshopper is an operating system explicitly designed to support orthogonal persistence. To this end, it provides three abstractions, *containers*, *loci* and *capabilities*, all of which are inherently persistent. *Containers* are the only storage abstraction provided by Grasshopper. They are conceptually very large address spaces capable of holding persistent data and are therefore suitable as coarse-grained persistent storage repositories. The data stored within a container may be directly referenced by a computation using an address relative to the start of the container. In contrast to address spaces in conventional operating systems, which are inextricably part of the *process* abstraction, containers can exist independently of computation. Thus, at certain times a container may be active and support multiple concurrent computations, while at other times it may be purely passive and void of activity.

The *locus* is the only abstraction over execution. Loci are scheduled by the kernel and execute within separate address spaces. The address space of a locus is composed from views of contiguous regions of various containers called *mappings*. In the simplest case, the address space of a locus contains a single mapping allowing it to access the contents of a particular container known as its *host*. Although each locus executes within a separate address space, a single container may host many loci allowing them to share the code and data stored within. Just as containers are not bound to any particular computation, the dual of this is true for loci. In other words, a locus is not bound to any particular container and may move to a new host container via the *invocation* mechanism [16]. Thus, containers and loci are completely orthogonal abstractions.

Figure 8 shows three loci and three containers. Each container stores code and data accessible to the loci executing within it. *Locus 1* and *Locus 2* are executing within *Container 1*, which is their current host. Hence, their address spaces map directly onto that of *Container 1*. Similarly, *Container 3* is hosting the execution of *Locus 3*. In contrast, *Container 2* is not currently hosting any loci and is completely passive.

The Grasshopper kernel must be able to control access to abstractions such as containers and loci. Furthermore, a naming mechanism is required to identify a particular container or locus to the kernel when performing system calls. Grasshopper provides the *capability* abstraction for this purpose [16]. A capability is conceptually a typed reference to an instance of a kernel abstraction combined with a set of access rights. The rights determine the operations that the holder of the capability may perform on the referend. Capabilities may be held by both containers and loci and are stored in a list associated with the holder. Capability lists are maintained within the kernel to prevent forgery.

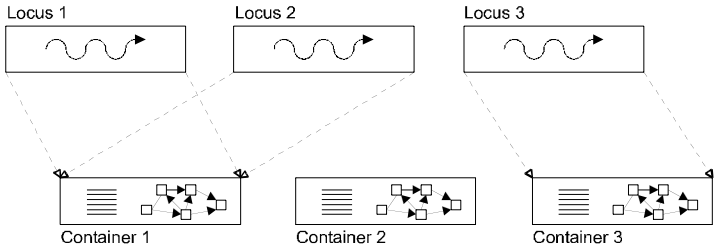

Figure 8: Pictorial representation of containers and loci.

2.6 Mapping

Mapping is a viewing mechanism used to compose address spaces from regions of other address spaces. In particular, mapping allows a contiguous region from one address space to be viewed from within a region of the same size in another address space. Changes made to data visible in either region are immediately reflected in both regions. Grasshopper provides two forms of mapping, *container mapping* and *locus mapping.* Container mapping allows the address space of a container to be composed from the address spaces of other containers. Since the address space of a locus is composed of regions mapped from the address spaces of various containers, any mappings affecting these containers will also be visible to the locus. For this reason, the effect of container mappings is said to be *global*. In contrast, locus mapping allows loci to install *private* mappings to container regions within their address spaces. These are typically used to provide access to per-locus data structures such as stacks. A more detailed account of the mapping mechanism and its possible uses is provided in [30].

2.7 Managing Persistent Data

From an external view of the Grasshopper system, it appears as if all data is stored within containers. In reality however, the actual data is maintained by *managers* [29]. A manager is a distinguished container holding code and data to support the transparent movement of data between primary and secondary storage. They are the only component of Grasshopper in which the distinction between long and short-lived data is apparent. To support the implementation of various different store architectures, managers have control over the virtual memory system in a similar manner to external pagers in micro-kernel operating systems such as Mach [3] and Chorus [12].

2.8 Problems with the Current Implementation

From our experience with the initial version of the Grasshopper system, we feel that we have made significant progress towards the goals we set out to achieve. However, through our porting efforts, a number of problem areas have been identified. Sections 2.8.1 through 2.8.5 describe the main problems and their origins.

2.8.1 Overhead of Page Fault Resolution

From past experience of building persistent object stores on top of Mach using the external pager interface [47, 48], it was decided that Grasshopper should support a similar mechanism to enable various store architectures to be tested without changing the kernel. Thus, the concept of a *manager* was developed to provide user-level control over the storage of persistent data. Since each container can have its own manager, many different store architectures can co-exist within the same system allowing each application to manage its persistent data in the most appropriate manner.

The problem with managers, and with external pager mechanisms in general, is that moving control over the virtual memory subsystem out of the kernel and into a user-level server adds to the latency with which page faults can be serviced. To illustrate this, consider the sequence of events required to service a page fault within a Grasshopper container.

1. Locus references a non-resident page, thereby inducing a trap into the kernel.
2. Kernel examines data structures to determine the container in which the fault occurred and hence the manager to notify.
3. Kernel upcalls into the manager.
4. Manager issues a system call to allocate a physical page.
5. Manager consults data structures to find the required page and issues a system call to read the page from disk into physical memory.
6. Manager issues a system call to bind the physical page into the address space of the faulting locus.
7. Manager returns control to the kernel.
8. Kernel resumes the faulting locus.

In the best case when the required page is already resident, it is necessary to call and return across the user-level/kernel boundary *three* times. Each time this occurs, certain registers must be saved and restored and capabilities must be checked to enforce protection. In the worst case when the page must be retrieved from disk, *five* boundary crosses are required. Although the cost of the disk I/O far outweighs the cost of the boundary crosses, it represents a significant overhead that could be better spent running other loci. In contrast, if page faults were serviced entirely within the kernel, the only boundary crosses would result from the initial traps, and since Grasshopper has a multi-threaded kernel, other loci could be scheduled during any necessary disk I/O.

2.8.2 Duplication of Information

Another source of inefficiency is the duplication of information within managers and the kernel. The most notable example of this involves the virtual address translation tables. Within the current implementation of the Grasshopper kernel, this information is effectively stored in three separate locations. Firstly, the information is held within the three-level hierarchical page tables (termed *contexts*) used by the memory management hardware. These contexts are typically sparsely populated making them expensive to maintain on a permanent basis. Therefore, a fixed-size pool of memory is used to cache the most recently used contexts.

Since contexts merely cache recently used address translations, a permanent record is maintained by the kernel using *local container descriptors* (LCDs) [28]. An LCD represents a mapping from container addresses to physical addresses and contains an entry for every resident page of the with which it is associated. Each LCD stores address translations very concisely within an extensible hash table. During the resolution of page faults, managers enter address translations into the appropriate LCDs via a system call and the kernel uses this information to create corresponding context entries when required. It was originally intended that the presence of an LCD entry for a particular page would stop the kernel from notifying the manager to resolve future faults on the same page. However, this policy prevents managers from using page faults to implement transactions. Therefore, the current implementation passes all faults on a page through to the manager. In light of this, managers must now

keep track of the current set of address translations. Although they can obtain this information using a system call to query the appropriate LCDs, it is more time-efficient if a separate hash table is maintained within the manager. This is crucial for implementing many of the common page replacement policies that operate by scanning the set of resident pages and examining their *dirty/referenced* status.

2.8.3 Context Switching Overhead

One of the fundamental aspects of Grasshopper is that a single container may host many loci simultaneously. When this occurs, the host container forms the basis for the address space of each locus. However, since a locus can privately map regions from other containers into its address space, it is necessary for loci to have separate address spaces. Restating this in another way, Grasshopper does not provide a way for two loci to share the same address space other than by arranging for them to share the same set of mappings. Thus, when context switching from one locus to another, the virtual address space must also be switched to that of the new locus. This is unfortunate since loci were intended to be extremely lightweight processes somewhat akin to threads, although in this implementation they are forced to use a heavyweight context switch much like conventional processes.

2.8.4 Suitability of Abstractions

The basic abstractions supported by Grasshopper were designed to bridge the semantic gap between the abstractions offered by conventional operating systems and those required by persistent application systems. Judging from our experience in porting applications to Grasshopper, we have largely succeeded in this goal. However, the chosen abstractions are not entirely appropriate in all areas. For example, the interface to disk devices uses physical memory to buffer I/O data. Therefore, any application wishing to work with disks must also deal with the complexity of managing physical memory. On the positive side, this makes disk I/O relatively efficient since the device driver can simply perform DMA using the physical buffers thereby eliminating the need to copy data to or from the appropriate virtual address. The downside is that applications making use of disks pay for the efficiency through added management complexity.

Another area in which the Grasshopper abstractions are inappropriate is the management of virtual memory as touched upon in Section 2.8.2. The duplication of information occurs partly because of the inefficiency of accessing the required information using the kernel interfaces provided. By providing a more direct interface to the hardware and kernel data structures, much of the duplication could be eliminated.

2.8.5 Complexity of the Kernel

Despite our best intentions, the current implementation of the Grasshopper kernel exhibits a monolithic architecture. The only system components residing outwith the kernel are the managers responsible for handling persistent data. The current implementation of the kernel contains code for each of the basic abstractions, all of the device drivers, the network protocols, and the checkpoint and recovery mechanisms for the internal meta-data. On top of this, the kernel is also multi-threaded and must consequently observe strict mutual exclusion conditions as well as protect itself against deadlock. All of this amounts to some 54,000 lines of C code. Big projects such as this are extremely difficult to maintain. Every time the kernel is altered there is a significant risk of introducing new errors, most of which are extremely subtle and invariably involve race conditions. This creates a disincentive to extend the kernel for fear of corrupting a working system.

In building a monolithic kernel, we are, in effect, guilty of creating a system similar to those that motivated the Grasshopper project in the first place. A key criticism of those systems was their inflexibility which stemmed from forcing applications to use the abstractions provided. We have improved a little on this through the introduction of managers to allow user-level control over persistent data. However, the remaining abstractions such as *loci*, *containers*, and *capabilities* are all fixed and may not be the most appropriate building blocks for all classes of application.

3. References

[1] *PS-algol Reference Manual - fourth edition*, . 1988, University of Glasgow and St Andrews.

[2] C.C.a.R.D. A Bensoussan, "The Multics Virtual Memory: Concepts and Design", *Communications of the ACM*, 15(5), pp. 308-318, 1972.

[3] M. Acceta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young, "Mach: A New Kernel Foundation for Unix Development", *Proceedings, Summer Usenix Conference*, , pp. 93-112, 1986.

[4] M.M. Astrahan, "System R: Relational Approach to Database Management", *ACM Transactions on Database Systems*, 1(2), pp. 97-137, 1976.

[5] M.P. Atkinson. "Programming Languages and Databases", in *Proceedings of Fourth IEEE International Conference on Very Large Databases*, pp. 408 - 419, 1978.

[6] M.P. Atkinson, K.J. Chisholm, and W.P. Cockshott, "CMS - A Chunk Management System", *Software Practice and Experience*, 13(3), pp. 259-272, 1983.

[7] A.L. Brown, "Persistent Object Stores", *Ph.D*, University of St. Andrews, http://www-fide.dcs.st-and.ac.uk/Info/Papers4.html#thesis.ab, 1988.

[8] A.L. Brown and W.P. Cockshott, *The CPOMS Persistent Object Management System*, . 1985, Universities of Glasgow and St Andrews.

[9] R.H. Campbell, G.M. Johnston, and V.F. Russo, "Choices (Class Hierarchical Open Interface for Custom Embedded Systems", *ACM Operating Systems Review*, 21(3), pp. 9-17, 1987.

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

[11] R. Chen and P. Dasgupta. "Linking Consistency with Object/Thread Semantics: An Approach to Robust Computation", in *Proceedings of Proceedings of the Ninth International Conference on Distributed Computing Systems*, pp. 1-17, 1989.

[12] Chorus-Systems, "Overview of the CHORUS Distributed Operating Systems", *Computer Systems - The Journal of the Usenix Association*, (Vol 1 No 4.), 1990.

[13] W.P. Cockshott, M.P. Atkinson, K.J. Chisholm, P.J. Bailey, and R. Morrison, "POMS: A Persistent Object Management System", *Software Practice and Experience*, 14(1), 1984.

[14] P. Dasgupta, R. Chen, *et al.*, "The Design and Implementation of the Clouds Distributed Operating System" *88/25*, Georgia Institute of Technology, 1988.

[15] p. Dasgupta, R. LeBlanc, A. Mustaque, and R. Umakishore, "The Clouds Distributed Operating System", *Technical Report 88/25*, Arizona State University, 1988.

[16] A. Dearle, R. di Bona, J. Farrow, F. Henskens, D. Hulse, A. Lindström, S. Norris, J. Rosenberg, and F. Vaughan. "Protection in the Grasshopper Operating System", in *Proceedings of Proceedings of the 6th International Workshop on Persistent Object Systems*, pp. 60-78, ftp://nezz.cs.stir.ac.uk/pub/papers/GH-04.ps.Z, 1994.

[17] A. Dearle, J. Rosenberg, F.A. Henskens, F.A. Vaughan, and K.J. Maciunas. "An Examination of Operating System Support for Persistent Object Systems", in *Proceedings of 25th Hawaii International Conference on System Sciences*, Poipu Beach, Kauaii, pp. 779-789, 1992.

[18] A. Dearle, J. Rosenberg, and F. Vaughan. "A Remote Execution Mechanism for Distributed Homogeneous Stable Stores", in *Proceedings of Third International Workshop on Database Programming Languages*, pp. 125-138, 1991.

[19] E. Elnozahy, D. Johnson, and W. Zwaenepoel. "The Performance of Consistent Checkpointing", in *Proceedings of 11th Symposium on Reliable Dsitributed Systems*, pp. 39-47, 1992.

[20] N. Hardy, "The Keykos Architecture", *Operating Systems Review, September, 1985*, , pp. 1-8, 1992.

[21] D.M. Harland, "REKURSIV: Object-oriented Computer Architecture", Ellis-Horwood Limited, 1988.

[22] F.A. Henskens, J. Rosenberg, and J.L. Keedy. "A Capability-based Distributed Shared Memory", in *Proceedings of Proceedings of the 14th Australian Computer Science Conference*, pp. 29.1-29.12, 1991.

[23] R. Jalili and F.A. Henskens. "Using Directed Graphs to Describe Entity Dependency in Stable Distributed Persistent Stores", in *Proceedings of Hawaii International Conference on System Sciences*, 1994.

[24] J.L. Keedy and J. Rosenberg. "Support for Objects in the MONADS Architecture", in *Proceedings of Proceedings of the 3rd International Workshop on Persistent Object Systems*, Persistent Object Systems, pp. 392-406, 1989.

[25] B. Koch, T. Schunke, A. Dearle, F. Vaughan, C. Marlin, R. Fazakerley, and C. Barter. "Cache Coherence and Storage Management in a Persistent Object System", in *Proceedings of Proceedings, The Fourth International Workshop on Persistent Object Systems*, pp. 99-109, 1990.

[26] K. Li, "Shared Virtual Memory on Loosely Coupled Multiprocessors", *Ph.D.*, Yale University, 1986.

[27] J. Liedtke. "On u-Kernel Construction", in *Proceedings of 15th ACM Symposium on Operating System Principles*, Copper Mountain Resort, Colorado, pp. 1-14, 1995.

[28] A. Lindström, "User level memory management and kernel persistence in the Grasshopper operating system", *Ph.D.*, University of Sydney, 1996.

[29] A. Lindstrom, A. Dearle, R. di Bona, J. Farrow, F. Henskens, J. Rosenberg, and F. Vaughan. "A Model For User-Level Memory Management in a Distributed, Persistent Environment", in *Proceedings of 17th Australian Computer Science Conference in*, Australian Computer Science Communications, pp. 343-354, http://nezz.cs.stir.ac.uk/~al/abstracts.html#GH-07, 1994.

[30] A. Lindstrom, J. Rosenberg, and A. Dearle. "The Grand Unified Theory of Address Spaces", in *Proceedings of Hot Topics in Operating Systems (HotOS-V)*, pp. 66-71, http://nezz.cs.stir.ac.uk/~al/abstracts.html#GH-11, 1995.

[31] R.A. Lorie, "Physical Integrity in a Large Segmented Database", *Association for Computing Machinery Transactions on Database Systems*, 2(1), pp. 91-104, 1977.

[32] R. Morrison, A.L. Brown, R.C.H. Connor, and A. Dearle, "The Napier88 Reference Manual", *Technical Report PPRR-77-89 URL: http://www-fide.dcs.st-and.ac.uk/Publications/1989.html#napier.reference.manual*, Universities of Glasgow and St Andrews, 1989.

[33] J.E.B. Moss, "Nested Transactions: An Approach to Distributed Computing", MIT Press, Cambridge, Mass, 1985.

[34] J.E.B. Moss, "Working with Persistent Objects: To Swizzle or Not to Swizzle", *IEEE Transactions on Computers*, , pp. 1-39, 1991.

[35] E.I. Organick, "The Multics System: An Examination of its Structure", MIT Press, Cambridge, Mass., 1972.

[36] R.D. Pose. "Capability Based, Tightly Coupled Multiprocessor Hardware to Support a Persistent Global Virtual Memory", in *Proceedings of Proceedings of the 22nd Annual Hawaii International Conference on System Sciences*, pp. 36-45, 1989.

[37] D.M. Ritchie and K. Thompson, "The UNIX Time-Sharing System", *The Bell System Technical Journal*, 63(6), pp. 1905-1930, 1978.

[38] J. Rosenberg. "The MONADS Architecture - A Layered View", in *Proceedings of Proceedings of the 4th International Workshop on Persistent Object Systems*, 1990.

[39] J. Rosenberg and D.A. Abramson. "MONADS-PC: A Capability Based Workstation to Support Software Engineering", in *Proceedings of 18th Hawaii International Conference on System Sciences*, pp. 515-522, 1985.

[40] J. Rosenberg, A. Dearle, D. Hulse, A. Lindström, and S. Norris, "Operating System Support for Persistent and Recoverable Computations", *CACM*, 39(9), pp. 62-69, 1996.

[41] J. Rosenberg, D. Koch, and J.L. Keedy, "A Massive Memory Supercomputer", *unknown*, , 1988.

[42] D.M. Ross, *Virtual Files: A Framework for Experimental Design*, . 1983, University of Edinburgh.

[43] M. Rozier, V. Abrossimov, *et al.*, "CHORUS Distributed Operating Systems", *Computing Systems*, 1(4), pp. 305-367, 1988.

[44] A.S. Tanenbaum, "Operating Systems: Design and Implementation", International Editions, Prentice Hall, 1987.

[45] S.M. Thatte. "Persistent Memory: A Storage Architecture for Object Oriented Database Systems", in *Proceedings of Proceedings of the ACM/IEEE International Workshop on Object-Oriented Database Systems*, pp. 148-159, 1986.

[46] I.L. Traiger, "Virtual Memory Management for Database Systems", *Operating Systems Review*, 16(4), pp. 26-48, 1982.

[47] F. Vaughan and A. Dearle. "Supporting Large Persistent Stores Using Conventional Hardware", in *Proceedings of 5th International Workshop on Persistent Object Systems*, pp. 34-53, 1992.

[48] F. Vaughan, T. Schunke, B. Koch, A. Dearle, C. Marlin, and C. Barter. "A Persistent Distributed Architecture Supported by the Mach Operating System", in *Proceedings of Proceedings of the 1st USENIX Conference on the Mach Operating System*, pp. 123-140, 1990.

[49] P. Wilson, "Pointer Swizzling at Page Fault Time: Efficiently Supporting Huge Address Spaces on Standard Hardware", *Computer Architecture News*, (June), pp. 6-13, 1991.