Open access • Book Chapter • DOI:10.1007/978-1-4471-3178-6_16

Stability in a Persistent Store Based on a Large Virtual Memory — Source link

John Rosenberg, Frans Henskens, Fred Brown, Ronald Morrison ...+1 more authors

Institutions: University of Newcastle, University of St Andrews

Published on: 01 Jan 1990

Topics: Memory management, Memory map, Computer memory, Virtual memory and Data diffusion machine

Related papers:

- Physical integrity in a large segmented database
- The Design and Implementation of a Log-structured file system
- Beating the I/O bottleneck: a case for log-structured file systems
- A fast file system for UNIX
- A case for redundant arrays of inexpensive disks (RAID)

Share this paper:

View more about this paper here: https://typeset.io/papers/stability-in-a-persistent-store-based-on-a-large-virtual-2dlqv5fp6r

This paper should be referenced as:

Rosenberg, J., Henskens, F., Brown, A.L., Morrison, R. & Munro, D. “Stability in a Persistent Store Based on a Large Virtual Memory”. In **Security and Persistence**, Rosenberg, J. & Keedy, J.L. (ed), Springer-Verlag (1990) pp 229-245.

Stability in a Persistent Store Based on a Large Virtual Memory

John Rosenberg and Frans Henskens
Department of Electrical Engineering & Computer Science
University of Newcastle
Australia

Fred Brown, Ron Morrison and David Munro
Department of Computational Science
University of St. Andrews
Scotland

ABSTRACT

Persistent systems support mechanisms which allow programs to create and manipulate arbitrary data structures which outlive the execution of the program which created them. A persistent store supports mechanisms for the storage and retrieval of objects in a uniform manner regardless of their lifetime. Since all data of the system is in this repository it is important that it always be in a consistent state. This property is called integrity. The integrity of the persistent store depends in part on the store being resilient to failures. That is, when an error occurs the store can recover to a previously recorded consistent state. The mechanism for recording this state and performing recovery is called stability. This paper considers an implementation of a persistent store based on a large virtual memory and shows how stability is achieved.

1. INTRODUCTION

Persistent systems support mechanisms which allow programs to create and manipulate arbitrary data structures which outlive the execution of the program which created them [6]. This has many advantages from both a software engineering and an efficiency viewpoint. In particular it removes the necessity for the programmer to *flatten* data structures in order to store them permanently. Such code for the conversion of data between an internal and external format has been claimed to typically constitute approximately thirty percent of most application systems [6]. In this sense a persistent system provides an alternative to a conventional file system for the storage of permanent data. This alternative is far more flexible in that both the data and its interrelationships can be stored in its original form. In order to achieve this a uniform storage abstraction is required. Such an abstraction is often called a *persistent store*. A persistent store supports mechanisms for the storage and retrieval of objects and their interrelationships in a uniform manner regardless of their lifetime.

Since all data of the system resides in the persistent store it is important that the integrity of the store be guaranteed, particularly after a system crash or hardware failure. The system must guarantee that, following a failure, the persistent store always returns to a consistent state as at some previous checkpoint. A store with this property is said to exhibit stability. We exclude here the question of total media failure which is a separate issue best handled by a backup or

dumping strategy. This requirement for a recovery mechanism is not peculiar to persistent systems. The same problem occurs with conventional file systems. For example UNIX provides limited recovery features. However, the problem is perhaps more acute with persistent systems. In a conventional file system each file is essentially an independent object. A loss of a single file following a crash is usually not a major problem. In a persistent system there may be arbitrary cross references between objects and thus a loss of an object can result in dangling references to the lost object. This may well compromise the integrity of the store. In this sense the problem of recovery within a persistent store is much more closely related to recovery in database systems [3].

A number of proposals for stable persistent stores have already appeared in the literature. The earliest of these [17] was oriented towards recovery in database systems and developed a new technique known as *shadow paging*. Many of the later systems are based on this technique [10, 23, 24, 25]. Several of these designs envisaged the use of hardware support which simplifies the implementation of shadow paging and results in a more efficient system [24, 25]. Ross [23] also implements shadow paging but utilises the facilities of VAX/VMS memory mapped files. Similarly, a system based on memory mapped files under SunOS 4 has been developed by Brown [10]. A different method of enquiry which is based on non-paged object mapping has been developed in other systems [5, 7, 9, 14].

In this paper we describe a stable persistent store based on a very large virtual memory. The implementation of this store is based on the MONADS-PC computer system [19], which has hardware support for large virtual address translation. We begin by describing the general principles of shadow paging as a technique for implementing stability. We then describe a previous implementation of the MONADS store without the recovery features. This is followed by a description of the implementation of shadow paging for MONADS-PC. It is shown that, given appropriate hardware, it is possible to implement a recovery scheme which minimises disk space overhead and allows considerable flexibility in maintaining consistency between multiple stores.

2. SHADOW PAGING - THE BASIC MECHANISM

Stability 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 [5, 7, 9, 10, 14, 17, 23, 24, 25]. The techniques differ in their efficiency with regard to the particular application area. However, there are two basic requirements. They are:

(a) the ability to perform an atomic update operation, and

(b) the ability to identify the old data and new data prior to the stabilise operation.

2.1 Atomic Update - Challis' Algorithm

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

(a) There exists 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.

(b) 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 will return to an efficient implementation later.

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 [11] 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. Figure 1 illustrates the state of the system prior to the n+1th stabilise operation. The root blocks record the two previous stabilised states n-1 and n.

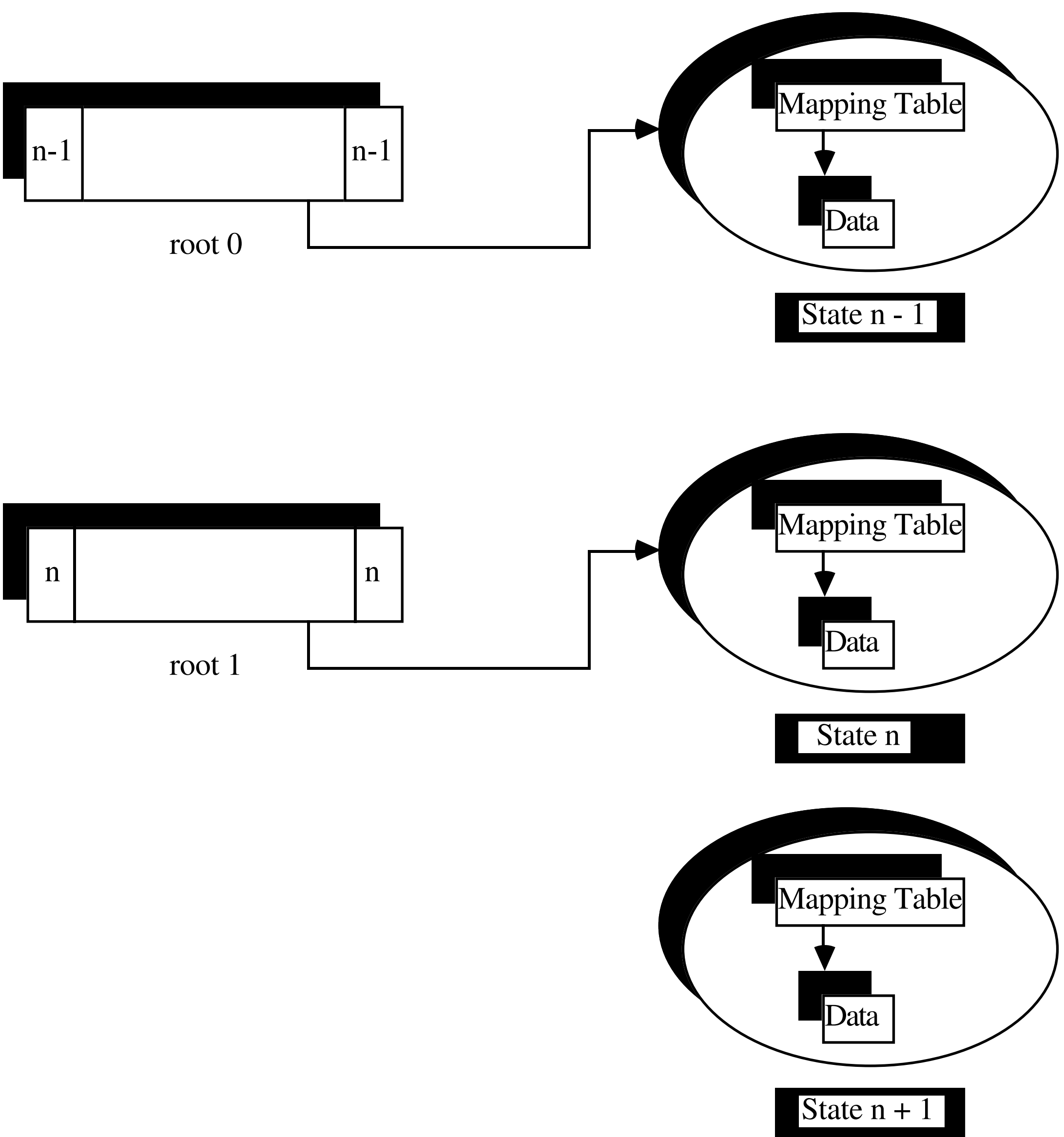

Figure 1: The state of the system prior to the n+1th stabilise operation

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 oldest version number, in this case n-1, with a new version number, n+1, and a pointer to the new updated mapping table. The space occupied by the old stabilised state n-1 may now be reused. This is illustrated in Figure 2.

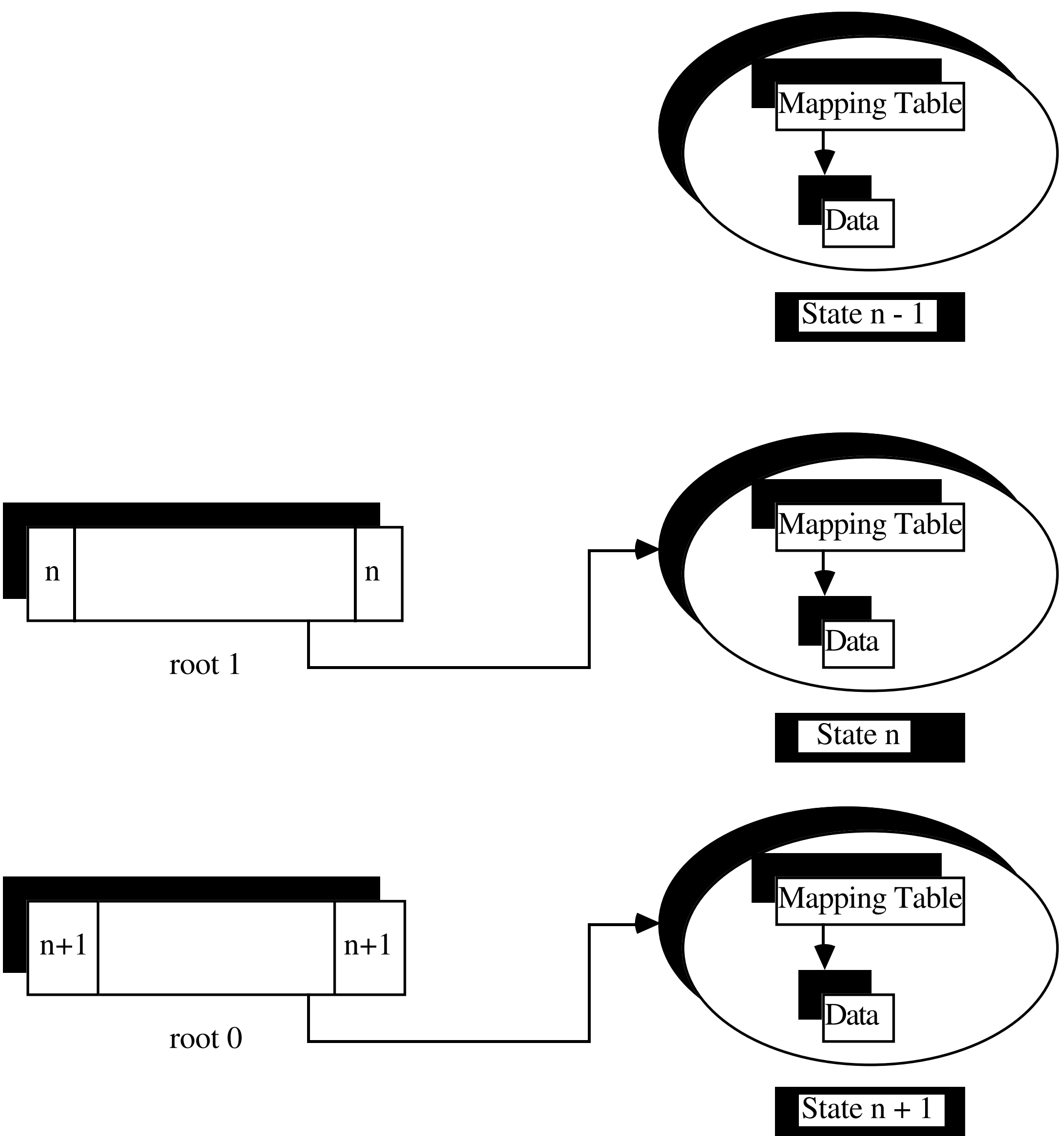

Figure 2: The state of the system after the n+1th stabilise operation

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.

2.2 A Paged Persistent Store With Shadowing

In a paged persistent store, the virtual address space is split up into pages of a fixed size which may be stored in disk blocks and main memory page frames of the same size. Where the virtual address space is not mapped in 1-to-1 correspondence with the physical disk, a table is required to perform the mapping. We call this table the disk page table.

The system operates as a paged virtual store using a page replacement algorithm for moving pages in and out of main store. The essence of the shadow paging technique is to ensure that when a modified page is written back to disk from main store that it is never written to the place it was read from. Instead a copy of the page, called the shadow page, is used to store the contents of the page between stabilise operations. By this means, only pages that have been modified have shadow copies and the system evolves incrementally.

The mechanism works as follows. On system startup the disk page table is copied into main store. From this all the pages of the persistent store may be found. We will refer to the stable version of the disk page table as the stable disk page table and to the copy in main store as the transient disk page table. All disk mapping is now performed using the transient disk page table. When a modified page is written back to disk, a shadow page is created, if it does not already exist, and the transient disk page table is altered to record the virtual address to shadow address mapping. On subsequent use of this page, the shadow address, which is found in the transient disk page table, is used.

A stabilise operation involves ensuring that all modified data is written back to disk, writing of the transient page table to disk and then performing an atomic update of the root block as described above. System startup and restore after a failure can also be performed as described above.

In reality, the operation of shadow paging is more complicated since the system must keep track of the free space on disk, detect a write operation on a page in main store and determine when a shadow page has already been created to avoid a shadow being created on every modified page discard.

In the above system, there are two copies of the disk page table. One on disk in a stable state and one in main store. The disk page table may, however, be too large to be keep in main store and would in that case be stored in a transient area of disk. This causes difficulties in the disk mapping as there is now a transient disk area which has to be mapped in some manner and the disk area mapped by the disk page table.

A solution to the above is to incorporate the disk page table in the virtual address space itself. This means that the page table is paged and that updates to it create shadow copies. The advantage of this is that there is only one paging mechanism and that changes to all data, including the page table, is incremental. The only requirement for such an organisation to work is that the address of the first page of the disk page table must be held at a fixed location within the virtual address space.

Finally, since the free space list must also undergo atomic update it can also be kept in the virtual address space.

The advantage of shadow paging is that only the pages that are altered require shadow copies and that the shadow copies are created incrementally on demand. This should be much cheaper than copying the whole page table or database. The disadvantage is that the paging system has

to be augmented. This is relatively cheap if performed at the design stage but may be quite difficult to retrofit to an existing system.

2.3 A More Detailed Account

We will now give a more detailed account of shadow paging by describing the total operation of the system.

As described in the previous section, the page table may be multi-level and is itself paged. The start address of the page table can always be found from the fixed position root block. In addition, the data contains a free space list for the disk. This must also be at an address known to the system within the logical address space. A shadow list is required to indicate that a shadow page has been created on disk and is being used. This list is not part of the virtual address space and is always considered transient. The final addition to the system is that each page frame requires a modified bit to indicate whether the corresponding page has been written on.

On system startup the information in the root block is brought into the main store. From this data all pages can be found on the disk. To evolve the system from one stable state to another efficiently, the technique makes changes incrementally, never overwriting the original data. Pages are only changed in main store and if they have been modified they are written out to a new disk block. Thus, updated pages, which may contain data, page tables or the free space list are duplicated incrementally, with the main store containing the volatile root of the updates. This volatile root will be different from the stable roots if any updates have been made.

There are six operations of the shadow paged store that concern us. They are:

(a) Create a new page.

To create a new page on disk the free space list is used to find a suitable disk block. The free space list is updated. The new page table is then altered to record the disk address given from the free space list and that the virtual address now has a valid mapping. The shadow list is updated to indicate that further copies do not need to be made if this page is subsequently written on. This operation may be recursive as the extending of the page table or the free space list may also require other create operations.

(b) Modify a page in main store.

The modify bit for the page frame is set when a page in main store is changed.

(c) Page fault.

On a page fault the system uses the new page table to find the address of the page on disk. This will be the shadow address if the page has already been modified.

(d) Page discard.

To discard a page from main store, the modify bit for the page frame is inspected. If it is clear the page frame can be overwritten without further action since a copy of the page exists on the disk. If not the shadow list is inspected. If a shadow exists the page can be written back to the address in the new page table which will be the shadow address. Otherwise a shadow page must be created. This involves finding a new disk block by using and updating the free space list and updating the page table to overwrite the entry for this virtual address. Again this may be recursive. The shadow list records the fact that a shadow has been created for this page to

avoid further shadowing on subsequent replacement. The page may now be written back using the new page table. Finally the main store modify bit is cleared for this page frame which is now available for reuse.

(e) Stabilise the persistent store.

To stabilise the persistent store, the system must first find all the pages in store that have been modified, since they have been altered but not yet written back to disk. If a shadow exists for these pages then they are written back immediately. Otherwise the system must create the shadow page as described above and write the page back to the shadow address given in the updated page table. An atomic update is then used to write the pointer to the new page table using the next system generated version number to the oldest root block. Finally, all modify bits in the main store and the shadow list are cleared. The system may now continue operation using the new version of the root.

The space occupied by the previous stable state can now be reused. This can be found by comparing the old and new page tables, performing a garbage collection to find the occupied disk blocks or by using a list of disk blocks which now have shadows that was constructed as the pages were altered.

(f) Restore the persistent store.

To restart the persistent store the action is as described in atomic update above.

3. THE MONADS VIRTUAL MEMORY

The MONADS persistent store is based on a paged uniform virtual memory. The virtual memory utilises large addresses which, in the current implementation are 60 bits. All storage, both temporary, i.e. RAM, and permanent, i.e. disk, is accessed via this virtual memory. The large virtual memory is divided into regions, called *address spaces*, each of which is up to 2^{28} bytes in size. Thus a virtual address has two components, a 32 bit address space number and a 28 bit offset within address space[1]. The address spaces are paged, using a four kilobyte page size.

3.1 Higher Level Architecture

Since all data of the system resides within the virtual store it is necessary to restrict the manner in which programs may construct a virtual address, in order to be able to implement a security policy. There seem to be two possibilities for implementing such a system. The first is to restrict code generation to certain trusted programs such as compilers. This is the approach taken in systems such as Napier88 [18]. An alternative scheme, adopted in MONADS, is to support a higher level architecture which provides mechanisms which can be used by compilers to implement various protection protocols, but which themselves guarantee that the ability to generate code cannot violate the security of the system. The higher level architecture supports arbitrary sized segments. Segment boundaries are orthogonal to page boundaries and thus segments may be as small as a few bytes or as large as an address space [15]. Segments are addressed by *segment capabilities* which contain the start address and length of the segment, as well as some access information, e.g. read-only. Segments may themselves contain segment capabilities, so that it is possible to build arbitrary data structures.

---

[1] A new implementation of the MONADS architecture known as MONADS-MM [21] has a 32 bit offset within address space and a 96 bit address space number. The main reason for the increase in address space number size is related to support for a local area network [2] and is not relevant to this paper.

In order to control access to the virtual store, segment capabilities are protected in such a way that they cannot be constructed or modified by programs. A suite of system management instructions is provided for creating and manipulating segments, and thus segment capabilities, in a controlled manner. A normal program cannot generate an arbitrary virtual address, but is constrained to address only those segments for which it has a segment capability. The kernel, which amongst other things is responsible for managing the virtual store, is provided with mechanisms to allow it to manufacture an arbitrary segment capability when required.

Segments are grouped together into *modules*. Each module resides in a separate address space and has a purely procedural interface. Access to a module is controlled by a *module capability*, which identifies the address space containing the data of a module and the type of access allowed in the form of a list of procedures. Module capabilities are protected by the architecture so that it is not possible to manufacture or modify them. If a module is deleted, then the subsequent use of a module capability referring to it will cause an exception to be raised. The full details of the higher level architecture are described elsewhere [20] and are beyond the scope of this paper.

3.2 Management of the Store

The disk store is divided into areas called *volumes*. A volume may be an entire physical disk or a logical partition of a disk. Each volume has a unique number. All pages of a given address space are held on the same volume and the corresponding volume number is encoded as part of the address space number. The most significant 6 bits are used in the current implementation[2] as shown in figure 3. Each time a new address space is created on a volume it is given a new unique within volume address space number. The numbers of deleted address spaces are never re-used. This means that it is not necessary to garbage collect across the entire store in order to reclaim unused addresses. The size of the address space number ensures that the system will never run out of addresses. In addition there is no need to remove references to deleted address spaces, since the use of such a reference will cause an exception to be raised.

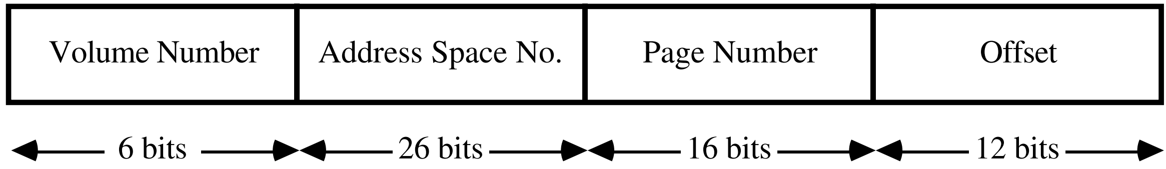

Figure 3: A MONADS virtual address

Access to a segment ultimately results in the generation of a virtual address. The data corresponding to that address may either be in memory or on disk, unless the address is invalid[3]. In conventional systems the page number portion of the address is used to index a page table which indicates whether the page is in memory or on disk and, in either case, indicates the address of the page as a disk block number or page frame address. This process is usually aided by an address translation buffer which caches recently used entries. The MONADS system uses a quite different technique which separates the task of translating virtual addresses for pages in memory from those for pages on disk. This is, in a sense, a return to the approach taken on the Atlas [16] and has many advantages in terms of efficiency and flexibility. These are discussed in [22].

[2]This is increased to 32 bits in MONADS-MM.

[3]At any point in time only some fraction of the virtual address space will be mapped to real store (either memory or disk). It is thus possible for an address to be generated which is invalid. This is most likely to happen if a deleted object is accessed since addresses are never re-used.

Translation of addresses for pages in main memory is achieved by the *address translation unit* (ATU). The ATU effectively simulates a very large associative memory and is implemented in dedicated fast memory as a hash table with imbedded overflow [1]. Similar schemes are described in [8, 13]. The hash table is large enough to hold an entry for every page frame of main memory. In fact it is several times larger to reduce the number of clashes. Thus the ATU, when given a virtual address, can either translate the address into a main memory address or indicate a page fault. The technique for maintaining the location of pages on disk is independent of the hardware and may even be different for separate address spaces. The ATU also supports read-only pages. An attempted write to such a page causes a write fault exception.

In the current implementation all address spaces are managed in the same manner. Each address space has associated with it a page table which holds the within volume disk addresses of each of its pages. This is called the *primary page table.* The maximum size of this page table is 2^{16} (2^{28-12}) entries or 2^{17} bytes, each entry being 16 bits. The primary page table is located at the highest addresses within the address space which it describes. A full size page table will occupy the last 32 pages. Note however that there may be gaps in the address space and pages of the page table are only created when required. That is, a page of the page table is created the first time one of the pages it references is created[4].

The key factor of this mechanism is that the page table is at a well defined address within the virtual address space and thus may be addressed using normal instructions. This considerably simplifies page fault resolution because no special mechanism need be used to access the page tables. On a page fault, the page fault handler is provided with the faulting virtual address. It then attempts to read the required page table entry, the address of which is easily generated since the address space number, the address of the page table within it and the faulting page number are known. If no page fault occurs when accessing the page table, then the disk address has been retrieved and the page fault is resolved. Otherwise the required page of the page table must be obtained. A separate *secondary page table*, or page table for the page table, is maintained for this purpose. It need only have 32 entries, one for each page of the page table, and it is held at a well defined address in page zero of each address space, along with some other red-tape information used by the higher level architecture. This is illustrated in figure 4. Thus, if a page fault occurs on the page table the address of the required secondary page table entry may be generated and the contents accessed.

The final problem to resolve is a page fault on page zero of an address space. Address space zero of each volume is used for this purpose and is called the *volume directory*. The volume directory has the same structure as other address spaces, with a primary and secondary page table. The data portion of the volume directory contains several data structures used by the system to manage the disk space. The two most important of these are the *hash table* and the *free space bit map.*

The hash table is a table containing the disk address of page zero of each valid address space on this volume. Given an address space number the hash table will either return the disk address of page zero or will fail. The latter indicates that the address space is invalid, either because it never existed or it has been deleted. Such an occurrence would cause an exception in the program causing the access. It is possible that a page fault occurs while accessing the hash table. However, this can be handled using exactly the same scheme as a normal page fault. Page zero of each volume directory, which contains the secondary page table for address space zero, is locked into main memory and so the process is always guaranteed to terminate. Page zero is called the *root page* and is located in a well known block of the disk. It is loaded into memory at system boot time, or whenever a new disk comes on-line.

---

[4]As an optimisation for small address spaces, the primary page table for the first one megabyte of an address space is actually held in the first page of the address space along with other red-tape information. The main primary page table is only created when the size of an address space exceeds one megabyte. This optimisation is ignored in the following discussion for simplicity.

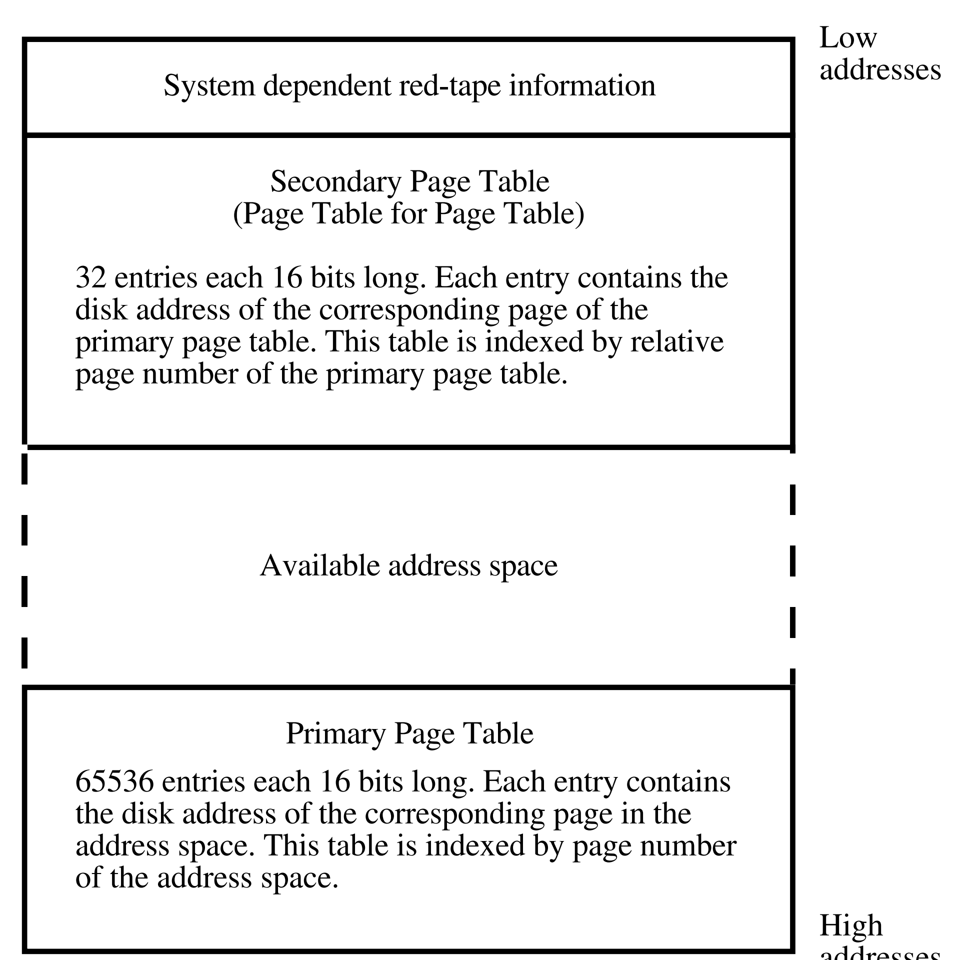

Figure 4: Address space structure

The free space bit map is a bit map indicating those disk blocks which are currently unallocated. In the current implementation the free space bit map is locked into main memory for each volume in order to simplify the page fault resolution process. This is not considered to be a serious deficiency since it is quite small, just over 8 Kbytes for a 256 Mbyte volume, and could be paged if necessary.

The final data structure worth mentioning is the *main memory table (MMT)*. The MMT is maintained on a global basis by the kernel and is locked into main memory. It contains one entry for each page frame of main memory. The MMT is used, amongst other things, for the allocation of page frames. Each entry indicates the corresponding virtual page number and the disk address of that virtual page, or that the page frame is currently unallocated. This is illustrated in figure 5. The disk address is used when a modified page is discarded in order to avoid accessing the page table again, since such an access may cause a page fault.

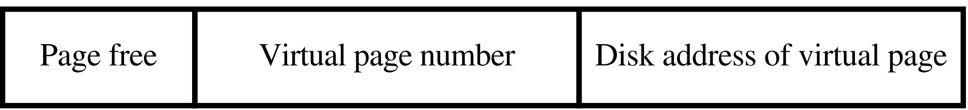

Figure 5: A *main memory table* entry

4. STABILITY IN MONADS

During the normal course of operation of the MONADS system integrity can be achieved by ensuring that all modified pages within the main memory are copied to the secondary storage medium before shutdown. However, this will not handle unexpected situations such as a system crash or a hardware failure. In these cases it is essential that, on restart, the system returns to a well defined state. In particular, it is essential that the data stored be in a consistent state. That is, all updates up to a given point in time are valid and none after that time.

In this section we describe an extension to the MONADS virtual memory scheme which achieves stability. The MONADS implementation is based heavily on Lorie's original shadow paging scheme and has many similarities at the address space level with that of Ross [23]. However, it is greatly simplified by the fact that all page tables, the hash table and the free space bit map themselves reside within the virtual store.

A major difference between the MONADS environment and other systems is that MONADS has its store partitioned into volumes, each of which is self-contained, in terms of space management. At this stage we will consider the stabilise operation to be based on individual volumes. This is an over-simplification since there are cross references between volumes and it is essential that they be kept consistent. We will also initially ignore the question of processes which are represented by process stacks held in the store. On a stabilise operation the state of such processes should be preserved so that they can be restarted following a crash. We will return to these issues later.

4.1 Single Volume Stabilise

In order to implement stability in MONADS an additional data structure is required. We will call this the *shadowed pages table (SPT)* and it will contain an entry for each page which has been modified since the last stabilise operation. This is required since it is necessary to detect the first modification to a page so that a shadow copy may be created. Although the ATU can detect an attempted write and does indicate modified pages, it only contains entries for pages currently in main memory. Consider a page which is modified and then discarded to disk. In this case a shadow page will be allocated. At a later stage the same page may be brought back into memory and again modified. The SPT will indicate in this case that a shadow page has already been allocated. We will see later that the SPT is also required to manage the release of disk space used by pages which are part of the previous checkpoint. At this stage we will consider the SPT to be a linear table, but will later look at alternative implementations. There is one SPT for each volume. The SPT contains two values for each entry. These are the disk address at the last checkpoint for the corresponding page, called the *old disk address*, and the disk address to which the page will be written at the next checkpoint, called the *new disk address*.

The other feature which we will employ is the ability of the ATU to support pages marked as read-only. On a write access to such a page an exception, called a *write-fault*, occurs. This is used to detect the first occasion on which a page is modified. This should not be confused with the *modify* bit in each entry of the ATU which indicates whether the corresponding page has been modified since it was brought into memory. The distinction is important for a situation where a page is marked as read/write immediately on being brought into the store, in which case the modify bit indicates if the page has been changed. This is used by the page fault handler to determine whether a discarded page needs to be copied to disk.

As described in section 3, the root page of a volume is effectively the root of a tree of disk addresses of pages on the volume. From the root page the disk address of any given page on that volume can be located. Following a checkpoint every page on the volume will either be in that tree or in the free space bit map, which itself is in a page described by that tree. The key to

implementing stability is to leave that tree undisturbed and to incrementally construct a new tree. This new tree can be pointed to by the in-memory copy of the root page, leaving the disk copy of the page pointing at the checkpoint version. Provided that none of the checkpoint pages is modified then, following a system crash, the system will return to the last checkpointed state without any processing being required. The new state described by the in-memory root page can be made the checkpointed state by a single disk write of the root page. In order to ensure that this write is atomic two root pages are maintained and Challis' algorithm, described earlier, is employed. Both of the root pages are placed at well known disk addresses so that they may be located at system start-up.

We now proceed to describe the rules for managing the shadow store, referring to the six operations described earlier. Note that some of these operations may result in subsequent faults. For example, a page fault may result in further page faults to retrieve a page table. These are handled recursively and each follows the procedure given below.

(a) To create a new page a new disk block is allocated using the free space bit map. If there are no free blocks then a stabilise must be initiated as described later. An entry for this page, containing a null old disk address and the new disk address, is added to the SPT. The new disk address is inserted into the MMT entry and the page table entry for the page is updated. If this is page zero of an address space then an entry is added to the volume hash table.

(b) The effect of modifying a page in main store depends on the read-only bit for the corresponding entry in the ATU. If the page is marked as read-write then the modify bit in the ATU is set and the access proceeds, otherwise a write fault exception occurs. In this case the original disk address of the faulting page is obtained from the MMT. A new disk block is allocated using the free space bit map. If there are no free blocks then a stabilise must be initiated as described later. An entry for this page containing the original disk address and the new disk address is added to the SPT. The MMT is updated with the new disk address and the page table entry for the page is updated. If this is page zero of an address space the hash table is updated with the new disk address.

(c) On a page fault the disk address of the required page is obtained and the page is read into a free page frame. This may involve resolving further page faults The disk address is looked up in the new address column of the SPT. If it is found then the page is mapped into the ATU as read-write, otherwise it is mapped in as read-only. In either case an entry containing the disk address of the page is added to the MMT.

(d) On a page discard, that is when a page is removed from memory, if the modify bit for the corresponding entry in the ATU is set then the page is written to the disk address indicated in the MMT. Note that if the page has been modified rules (a), (b) and (c) will guarantee that a new disk block has already been allocated.

(e) A *stabilise* operation for a volume may either be automatically generated or explicitly requested by a user/program. In either case the following must be performed. For each entry in the SPT the old disk address is extracted and the corresponding bit in the free space bit map is set. We will assume that the free space bit map is locked into main memory and therefore cannot cause a page fault. All pages for this volume in memory which have been modified are copied back to disk. These can be easily located using the MMT and the ATU. Note that the order in which these are written to disk is not important since if there was to be a system crash, the old state described by the old root page on disk would be restored and thus all of the blocks allocated as part of the new state would be in the free space bit map. These pages should then be marked as read-only in the ATU so that, if they are subsequently modified, a new disk block may be allocated. Finally the root page for the volume is written back to disk. This final step makes the new state the stable state. At this point the SPT is cleared and the checkpoint is complete.

(f) A *restore* operation takes place following a crash of a volume. Since the entire state as at the last checkpoint still exists on secondary storage, and all disk blocks used since that checkpoint will still be in the free list of that checkpoint state, no modification to the secondary store needs to be performed in order to restore to the last checkpoint. The SPT for the volume is cleared and any pages from that volume in main memory must be removed from the ATU. The root page from the last checkpoint is then retrieved and system operation may continue.

Notice that in this scheme a new disk block is allocated *before* a page is modified. This guarantees that it is always possible to stabilise, that is there is always sufficient disk space. Given the page table structure it is possible for the page fault handler to statically calculate the maximum number of pages which may be modified as a result of processing a write fault and to ensure that there is sufficient disk space for each of these pages before granting write access to the page. This number of pages never exceeds five in the MONADS scheme.

We now return to the question of implementation of the SPT. There are three operations which must be performed on the SPT. These are insert a new entry, check if an entry with a particular new disk address is in the table and cycle through each entry in the table. We can suggest two alternative implementations. The first is a hash table, using selected bits of the new disk address as the hash key. This would provide good performance on all of the required operations. However, the size of the table is a potential problem since, in theory, it can grow quite large, with entries for half the number of disk blocks on the volume. However, in practice this is not likely to be a problem since it is sensible to checkpoint frequently. In any case, if the table became full a checkpoint could be forced. Since a checkpoint can be performed at any time it is possible for the system to enforce a policy on checkpoints to avoid this situation, e.g. checkpoint after *n* pages have been modified.

An alternative implementation of the SPT is to use two bit lists, both of which have one bit for each disk block on the volume. For the maximum size MONADS volume, 256 megabytes, this is only 8 kilobytes each, which it is feasible to lock into main memory. The bit lists effectively correspond to the two columns of the SPT, the first indicating the old disk addresses of modified pages and the second indicating the new disk addresses of modified pages. This allows the three required operations to be performed efficiently and in a fixed amount of store. In fact these bit lists operate in a similar manner to Lorie's MAP and shadow bits, but have the advantage that, since they are not in the page tables, they may easily be cleared following a stabilise operation.

4.2 Multi-volume Consistency and Processes

We indicated earlier that, in the MONADS system, it is possible to have cross references between address spaces on different volumes. Independent volume checkpointing in this circumstance could result in inconsistencies following a crash. A solution to this is to implement a multi-volume stabilise using a two-phase commit. This would rely on making one of the volumes, presumably on a fixed disk, a master volume. The master volume would record which root block to use on each dependent volume. Two copies of each root page are maintained as before. Each volume would be stabilised as above, but only the older of the two root pages is updated with the new timestamp. The master volume is updated last, with both root pages being written to guarantee that the write is successful. Following a crash the timestamps can be inspected to determine the most recent consistent state.

Processes can be included in such a scheme by saving the current state of each process, including the contents of screen buffers, etc., before commencing the checkpoint operation. At restart this state information can be retrieved and the processes continued. The process state information could either be saved on the individual process stacks or in a central object pointed to by the master volume.

A potential disadvantage of this scheme is that the entire store must stabilised at one time. For a large configuration with many volumes this could become quite expensive since all processes must be stopped during the stabilise. However, the situation is not be as bad as it at first seems. Much of the work takes place in parallel with the normal operation of the system as part of the page discard task and at most the entire memory of the machine, but usually much less, must be copied to disk at a checkpoint.

This scheme can be generalised to allow for a very flexible stable store in which volumes are stabilised in groups in such a way that the groupings may be changed as required. For example, it may be that a particular volume supports a self-contained related group of users and their data. In such a case that volume could be stabilised by itself. Given an appropriate mechanism it would then be possible to group that volume with another so that the two are stabilised together. As another example consider bringing a volume from another site and mounting it on a machine. It may be desirable for it to be stabilised with other volumes on that machine. This can easily be achieved by the proposed scheme.

5. DISCUSSION

We will now consider the effects of the above scheme on the placement of pages on the disk. After some period of time the pages of an address space may be randomly distributed across the disk. This is acceptable if the pages are to be randomly accessed. However, for sequential access it would be better if the pages were physically sequential. This was achieved in Brown's scheme [10] by creating a pre-copy of pages on disk and overwriting the original page in place, maintaining the original physical structure of the store. This has two disadvantages. First, each modified page must be physically copied and second, the store must be partitioned into two areas, store and shadow pages, potentially reducing the disk space utilisation. However, it is desirable to support efficient sequential access.

Lorie [17] has suggested a solution to this problem. The disk, or volume in our case, is organised into physical clusters. Each cluster consists of a set of disk blocks such that the head movement time between blocks in the same cluster is much smaller than the head movement time between blocks in different clusters. Each address space is associated with a cluster and when a new disk page is required for an address space it is allocated in this cluster, if possible. By careful choice of the cluster size it should be possible to achieve good locality for sequential access. Clustering need not be implemented globally, but can be an option on an address space basis.

In several of the schemes described in the literature [10, 24] there is a disk space overhead, even following a checkpoint operation. In the proposed scheme disk space is allocated fully dynamically. There is no static division of store into shadow and main store and once a stabilise has taken place all shadow store is immediately released. By implementing a clustering scheme as described above, this improvement can be achieved without serious performance degradation for sequential access.

There is an overhead in terms of both disk space and execution time in performing the shadow paging algorithm. It is quite likely that for certain address spaces containing temporary objects stability is of no importance. In these cases it is desirable to disable the shadowing. The MONADS scheme can be enhanced to support this option. Each address space can be flagged as either shadowed or non-shadowed. In the latter case the page table and red-tape information would still be shadowed, but not the data. This is required in order to ensure the integrity of store management data.

6. CONCLUSION

Persistent systems have the potential to provide a powerful and flexible software development environment. However, if they are to achieve that goal they must be both efficient and robust. We have addressed the former issue by providing purpose-built hardware specifically designed to support a large virtual store. In this paper we have suggested a scheme to make this virtual store stable.

The scheme is based on shadow paging but has the advantage that disk space allocation is fully dynamic. A minimum of disk space is used. At any time there are at most two copies of any page on disk, the last checkpoint version and the current version if the page has been modified. Following a checkpoint, there is only one copy of each page. At no time do pages have to be copied, either in memory or on disk. Following a system crash the system automatically returns to the last consistent state with no post-processing of the disks. The checkpoint process can be expensive but much of the work may be overlapped with the normal operation of the system.

The scheme gains simplicity through two techniques. The first is the maintenance of all of the virtual memory tables, free space bit maps, etc. within the store. This allows the shadowing technique to be used recursively on the page tables themselves, considerably simplifying the implementation. The second technique is the use of very large addresses. This allows the virtual address space to be partitioned to support multiple volumes, without fragmenting the primary or secondary store.

An interesting area for further research is the use of an *uninterruptable power supply (UPS)*. Technology in this area has improved considerably and it is now quite possible to provide battery backup to ensure maintenance of power to disks and memory for an extended time, at least in the order of hours [12]. Given this sort of technology the question of coping with power failure is no longer an issue. Following a power failure all data can simply be copied to disk. However, this does not cope with the situation of a system software failure or, even more seriously a hardware failure (e.g. processor error) where the power to processor and memory must be removed in order to rectify the fault. These situations will still require another mechanism such as stability. However, we are investigating possible simplifications and improvements to the proposed mechanism based on the use of a UPS. In particular it should be possible to considerably reduce the I/O overheads by shadowing within main memory.

ACKNOWLEDGEMENTS

The authors wish to thank Peter Broessler for reading several earlier drafts of this paper and making many helpful suggestions. This work was undertaken during John Rosenberg's study leave period at the University of St Andrews and was supported by SERC grant GR/F 28571.

REFERENCES

1. Abramson, D.A. "Hardware Management of a Large Virtual Memory", *Proceedings 4th Australian Computer Science Conference*, Brisbane 1981, pp. 1-13.

2. Abramson, D.A. and Keedy, J.L. "Implementing a Large Virtual Memory in a Distributed Computing System", *Proceedings of 18th Annual Hawaii International Conference on System Sciences*, 1985, pp. 515-522.

3. Astrahan, M.M. et al "System R: Relational Approach to Database Management", *ACM Transactions on Database Systems*, 1, 2, June 1976, pp. 97-137.

4. Atkinson, M.P., Chisholm, K.J. and Cockshott, W.P. "PS-algol: An Algol with a Persistent Heap", *ACM SIGPLAN Notices*, 17, 7, July 1981, pp. 24-31.

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

6. Atkinson, M.P., Bailey, P., Chisholm, K.J., Cockshott, W.P. and Morrison, R. "An Approach to Persistent Programming", *The Computer Journal*, 26, 4, Nov. 1983, pp 360-365.

7. Atkinson, M.P., Bailey, P.J., Cockshott, W.P., Chisholm, K.J. and Morrison, R. "POMS: A Persistent Object Management System", Software Practice and Experience, 14, 1, January 1984, pp. 49-71.

8. Berstis, V., Truxal, C.D. and Ranweiler, J.G. "System/38 Addressing and Authorization", IBM System/38 Technical Developments, 1978, pp. 51-54.

9. Brown, A.L. and Cockshott, W.P. "The CPOMS Persistent Object Management System", Universities of Glasgow and St Andrews PPRR-13, Scotland 1985.

10. Brown, A.L. "Persistent Object Stores", Ph.D. thesis, available as Persistent Programming Report 71, 1989, Universities of St. Andrews and Glasgow.

11. Challis, M.F. "Database Consistency and Integrity in a Multi-user Environment", in *Databases: Improving Usability and Responsiveness*, B. Schneiderman (editor), Academic Press 1978, pp. 245-270.

12. Copeland, G., Keller, T., Krishnamurthy, R. and Smith. M. "The Case for Safe RAM", *Proceedings of the 15th International Conference on Very Large Databases*, Amsterdam 1989, pp. 327-335.

13. Edwards, D.B.E., Knowles, A.E. and Woods, J.V. "MU6-G: A New Design to Achieve Mainframe Performance from a Mini-sized Computer", *Proceedings of the 7th Annual Symposium on Computer Architecture*, Computer Architecture News, 8, 3, May 1980, pp. 161-167.

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

15. Keedy, J.L. "Paging and Small Segments: A Memory Management Model", *Proceedings 8th World Computer Congress (IFIP-80)*, Melbourne 1980, pp. 337-342.

16. Kilburn, T., Edwards, D., Lanigan, M. and Sumner, F. "One Level Storage System", *IEEE Transactions*, EC-11, 2, 1962.

17. Lorie, R.A. "Physical Integrity in a Large Segmented Database", *ACM Transactions on Database Systems*, 2, 1, March 1977, pp. 91-104.

18. Morrison, R., Brown, A.L., Carrick, R., Connor, R., Dearle, A. and Atkinson, M.P. "The Napier Type System", Proceedings of the 3rd International Workshop on Persistent Object Systems, Newcastle, 1989.

19. Rosenberg, J. and Abramson, D.A. "A Capability-Based Workstation to Support Software Engineering", *Proceedings of 18th Annual Hawaii International Conference on System Sciences*, 1985, pp. 222-230.

20. Rosenberg, J.L. and Keedy, J.L. "Object Management and Addressing in the MONADS Architecture", *Proceedings 2nd International Workshop on Persistent Object Systems*, Appin Scotland, 1987, available as PPRR-44, Universities of Glasgow and St. Andrews.

21. Rosenberg, J., Koch, D.M. and Keedy, J.L. "A Massive Memory Supercomputer", *Proceedings of 22nd Annual Hawaii International Conference on System Sciences*, 1989, pp. 338-345.

22. Rosenberg, J., Keedy, J.L. and Abramson, D.A. "Addressing Mechanisms for Large Virtual Memories", Research Report CS/90/2, University of St. Andrews, 1990.

23. Ross, D.M. "Virtual Files: A Framework for Experimental Design", Department of Computer Science, University of Edinburgh, CST-26-83, October 1983.

24. Thatte, S.M. "Persistent Memory", *Proceedings of IEEE Workshop on Object-Oriented DBMS*, 1986, pp. 148-159.

25. Traiger, I.L. "Virtual Memory Management for Database Systems", *Operating Systems Review*, 16, 4, October 1982, pp. 26-48.