Architectural and Operating System Support for Orthogonal Persistence

John Rosenberg University of Sydney, Australia

ABSTRACT: Over the past ten years much research effort has been expended in attempting to build systems which support orthogonal persistence. Such systems allow all data to persist for an arbitrary length of time, possibly longer than the creating program, and support access and manipulation of data in a uniform manner, regardless of how long it persists. Persistent systems are usually based on a persistent store which provides storage for objects. Most existing persistent systems have been developed above conventional architectures and/or operating systems. In this paper we argue that conventional architectures provide an inappropriate base for persistent object systems and that we must look towards new architectures if we are to achieve acceptable performance. The examples given are based on the Monads architecture which provides explicit hardware support for persistence and objects.

1. Introduction

Over the past ten years much research effort has been expended in attempting to build systems which support orthogonal persistence [1, 2, 4, 5, 6, 7]. The idea behind persistence [12] is simple: that 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 types may be persistent and that data may be manipulated in a uniform manner regardless of the length of time it persists. Orthogonal persistence is not found in contemporary operating systems, nor 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 persistence have been constructed [3, 10, 25, 51]. These usually provide a large store within which concurrent processes may manipulate persistent data. In the more advanced of these systems, the stores contain all data including procedures, graphics objects, processes and their associated state. Perhaps the most important feature of persistent systems is that this store is both resilient and stable. Persistent systems that have been constructed to date, with a few exceptions [23, 53], have been constructed on top of traditional operating systems and conventional architectures. In addition these systems have concentrated mainly on language issues and it is only recently that there has been more interest in providing support at a lower level.

It is common to find a notion of *objects* associated with persistent systems. That is the persistent store is an object store, containing objects which have an associated type. Usually this store is maintained in a type secure fashion, so that objects can only be accessed in a manner appropriate to their type, although this is difficult in a multi-lingual

system. Most persistent systems do not support inheritance, although Connor has been investigating a generalised subtyping facility for the persistent language Napier88 [22].

A further area of interest is security in the sense of control of access to data, as distinct from the protection provided by a type system. In a conventional system security and protection are provided by two mechanisms. The first mechanism is support for some form of separate per process address space which is usually implemented using hardware assistance. This restricts access to data in virtual memory on a coarse grain basis. The second form of security is provided by the file system which controls access to long term data. In a persistent system there is no file system; instead all data is stored in the persistent store. It is therefore essential to provide alternate security mechanisms to allow users to control access to their data. This will be further discussed later.

In this paper we review the support required for persistent object systems. It is shown that considerable architectural and operating system support is required in order to construct scalable and efficient persistent object systems. The paper concludes with a brief description of a new machine specifically designed to support persistent object systems and some suggestions for the requirements of future persistent systems.

2. Background

To place the remainder of the paper into an appropriate context, in this section we provide some background on persistence. A definition of orthogonal persistence is given and we define the use of the term object in the context of existing persistent systems.

2.1 Persistence

Atkinson et al [12] have identified three principles which underly the concept of orthogonal persistence. They are:

- *Persistence independence*. The manner in which data is created and manipulated is independent of the time for which that data

persists. Programmers have no control over the transfer of data between long and short term storage; such transfers are performed transparently by the system itself.

- *Data type orthogonality*. There are no types of object for which special cases apply; all types (including code) are allowed the full range of persistence.
- *Persistence identification*. The mechanism for providing and identifying persistent objects is orthogonal to the type system, computational model and control structures.

Unfortunately many so-called persistent systems fail on one or more of these criteria. All persistent systems support the first principle by definition. Some persistent system designers have taken a pragmatic approach and have only allowed particular types to persist in violation of the second principle. For example, Pascal/R [55] only allows first order relations to persist, and then only if they do not contain pointers. This creates a discontinuity in the system, makes it more difficult to understand and means that applications which do not fit into the relational model cannot be implemented without resorting to an external file system.

In many persistent systems persistence is determined by reachability from some form of persistent root [13, 15, 19]. Thus all persistent objects can be found by computing the transitive closure of that root. Other systems have taken the approach of associating persistence with type [40, 50] in violation of the third principle (and the second as a consequence). This can result in dangling references, or at best invalidated fields of structures, as shown in Figures 1 and 2 from [43].

Figure 1 shows a structured object containing pointers to two persistent objects and one non-persistent object before being copied to the persistent store. Since the non-persistent value cannot be saved in the store the copy operation results in the object shown in Figure 2. Such inconsistencies make it more difficult to build systems and result in latent errors. Systems which support persistence by reachability never have this problem and are therefore preferable.

There are several advantages of persistent systems [43]. First they reduce complexity for application builders. In conventional systems the system designer must map the real-world onto a combination of a programming language and a database/file system. These two mappings often have complex interactions and may be somewhat incompatible.

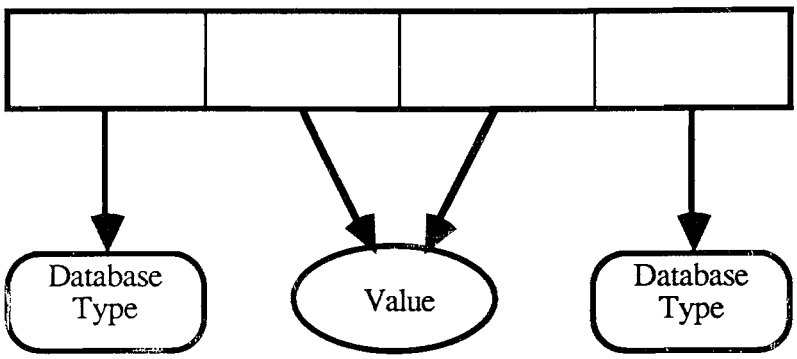

Figure 1: Persistent objects before being sent to the store.

A second advantage is the potential reduction in code size and execution time. It has been suggested that approximately thirty percent of the code in typical database applications is solely responsible for the mapping between the internal (temporary) and the external (persistent) format [12]. This usually involves "flattening" and "rebuilding" data structures and copying data to and from files. Such explicit code is unnecessary in a persistent system since all data is maintained in a common format and copying to and from permanent storage is performed invisibly by the system. This results in considerable savings during applications development and a reduction in total execution time.

Third, since all data resides in the one store, a single model of protection may be employed. This may be based purely on type security or may be hardware supported in order to provide multi-lingual support [45]. In either case the user need only be concerned with the

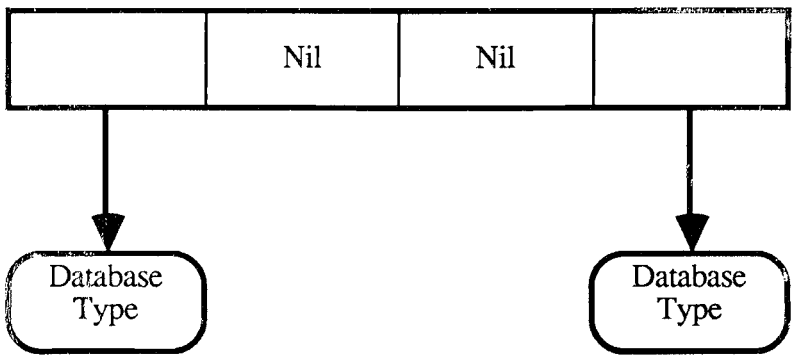

Figure 2: The same objects as in Figure 1 in the persistent store.

one mechanism as distinct from the multi-level protection involving processes and files on conventional systems.

Of course, the advantages of orthogonal persistence are not achieved without a cost. In section 3 of the paper we will examine the impact of persistence on memory management, protection and distribution.

2.2 Objects

As we have suggested in the introduction, persistent systems usually support a notion of objects. These objects are not in general the encapsulated abstract data types supported by most object-oriented languages. They are simply typed objects, including scalars, structures and arrays which may stored and retrieved in a type secure manner. Higher level structures can be built by using mechanisms provided by the programming language. Some persistent systems do directly support encapsulated objects (e.g. Napier88 [44] and Monads [51]). However, most do not (as yet) support any notion of inheritance.

It is interesting to reflect on the differences between conventional object-oriented languages and a persistent object-oriented language. In a language such as C++ [56] support for objects and encapsulation can effectively be provided by the compiler. The compiler associates procedures (methods) with the encapsulated data and supports method invocation, perhaps with the assistance of a run-time library. In a sense the encapsulation is "compiled away" and at run-time the object-oriented program becomes a single monolithic program in the same sense as a program written in a conventional language.

The situation in a persistent object system is more complex. Since objects are allowed to persist beyond the length of the program which created them the encapsulation must be maintained at run-time and on long-term storage so that the methods, the data and the interrelationships are preserved. In this way the system can guarantee the encapsulation and type security of objects.

3. Support for Persistent Objects

In this section we examine architectural support required for the efficient implementation of persistent object systems. The emphasis is on flexibility and efficiency. Persistent systems support a radically differ-

ent paradigm from conventional systems and it is unreasonable to expect existing architectures and operating systems to provide an appropriate basis for their development. The desirable characteristics of systems to support persistence are examined in terms of memory management, protection and distribution.

3.1 Memory Management

In a conventional computer system the storage connected to a machine is divided into two distinct regions. Active code and data is directly addressable by machine instructions and is usually held in main (or virtual) memory. A hardware supported address space of some specific size (usually up to 4 gigabytes, but maybe larger, e.g. AS400, R4000) is provided and hardware assisted mechanisms translate the addresses used by programs into main memory addresses. On most modern machines a virtual memory is supported, whereby not all of the active code and data is held in real memory; some may be held on a dedicated area of a secondary storage device, usually called the swap area. The address translation hardware uses tables to indicate which sections of the code and data are in memory and which are not. If an attempt is made to access data not in memory a fault occurs and system software loads the required data. This mechanism allows the transparent extension of the address space beyond the size of main memory. Notice, that it is only used for temporary code and data and all data held in the swap area ceases to exist when the program terminates.

Permanent data is held on secondary storage devices in the file store and is accessed using a quite separate mechanism in the form of a file system. There is usually no direct hardware support for the file system (other than the ability to access storage devices). On the other hand, much larger volumes of storage are provided by the file system. Storage in the order of 10 to 20 gigabytes would not be unusual.

In a persistent system all storage is treated in a uniform manner and it is therefore desirable that there be a uniform addressing mechanism. The obvious approach is to extend the notion of virtual memory to encompass all storage. This approach has been adopted on several persistent systems [17, 20, 51]. The major difficulty lies in the size of addresses. Since all storage must be addressed, the address size must be significantly larger than that required for conventional systems. In addition, in order to minimise the frequency of garbage collection it

may be desirable to allow "holes" in the address space, further increasing the required address size. Finally, some protection paradigms require that addresses not be re-used, thus forcing a further increase in address size. For all of these reasons addresses in the order of 64 to 128 bits may be desirable.

We are left then with the problem of efficiently supporting a very large address space. It is also interesting to note that the nature of persistent systems is such that accesses to data within this address space may be scattered in an essentially random fashion over a large range of addresses. For example, traversing a large persistent data structure will involve a single access to each of a large number of objects. Thus there may well be less locality in the references than there would be in a conventional machine. The key issue is providing an efficient address translation mechanism for large virtual addresses.

It has been suggested that such address translation can be provided by software and this has become known as *pointer swizzling*. The idea is that objects held on secondary storage use large addresses and these are mapped onto smaller machine-supported virtual addresses as the objects are copied into memory. For example, the Napier88 system maintains a local heap and all objects are swizzled as they are brought into this heap [16]. More recently Wilson [61] has described an alternative technique via which all objects in a given page are swizzled at the time the page is brought into memory. This technique can utilise the virtual memory hardware to detect accesses to non-swizzled objects. While these techniques do work, they incur considerable overheads. For example, Wilson's scheme requires that all pages be scanned as they are brought into memory and there are considerable costs during page discard.

For this reason a number of research groups have been exploring techniques for providing architectural support for large address translation. Two basic approaches have been taken. The first is based on a paged address space with objects mapped onto pages, and the second employs object-based address translation. We will consider each of these in turn.

In systems which support objects mapped onto a paged address space each object is simply allocated a contiguous region of the address space. The region may overlap page boundaries in an arbitrary fashion. The object is given an identity which corresponds to its starting virtual address. On each object access the object identity and an

offset are provided and these may simply be added (possibly by an index addressing mode) to generate the direct virtual address of the required data. If a page fault occurs then this may be handled in the usual manner.

Most modern machines employ a hardware-assisted paged address translation scheme with the entire address space being divided into fixed size pages, typically about 4 kilobytes long. A page table is maintained which maps virtual page numbers to physical page frame numbers. Each entry contains a presence bit and the page frame number. For pages which are currently not in main memory the entry holds the disk address of the page. Logically on each access the virtual page number is used as an index into the page table and the page frame number is obtained or a page fault occurs. In practice a dedicated cache of address translation entries (called a translation lookaside buffer or TLB) is used to improve performance. The TLB is usually maintained on a least recently used basis. Note that the TLB relies heavily on locality of reference to ensure that a high percentage of the required address translation entries will be found in the TLB.

In the scheme described above the page tables are used for two purposes. First to hold the data for translation from virtual to main memory addresses and second to hold data for translation from virtual to secondary storage addresses. This dual usage restricts flexibility for both of these translation mechanisms. In particular the page tables become extremely large and difficult to manage as the address space grows. It is possible to partly divorce these two with conventional hardware and this has been done in systems such as Mach [9] and Chorus [18] which support machine independent virtual memory management.

A quite different approach to paging has been adopted in the Monads architecture [51], which is explicitly designed to support a very large address space. A hardware-supported mechanism for translating from virtual addresses to main memory addresses is provided. This may be viewed as a black box which either translates a virtual address (if the page is in memory) or causes a page fault.The scheme uses a hash table, with embedded overflow to resolve synonyms. Hash tables have been used on other machines to support address translation. The MU6-G [28] used multiple hash tables searched in parallel and the IBM System/38 [14] used a single hash table held in main memory. Both of these machines also had a dedicated address translation cache

to improve performance. In our scheme the hash table is entirely held in dedicated high speed memory. Each cell of the table contains a key field identifying the virtual page, a main memory page frame number, a link field, a read-only bit and various status flags used to control the translation process (see Figure 3).

When presented with a virtual address the hardware hashes the address to obtain a cell address within the table. The key field in the cell is compared with the original virtual address, and if there is a match, the main memory page frame number is used to form a main memory address. If there is a mismatch, the link field is used to follow a chain of synonyms. The chain is terminated by an end of chain status bit. If a virtual page is not found in the hash table then a page fault is generated.

Insertion and deletion operations are only performed when a page is loaded into or discarded from main memory. The operations are not

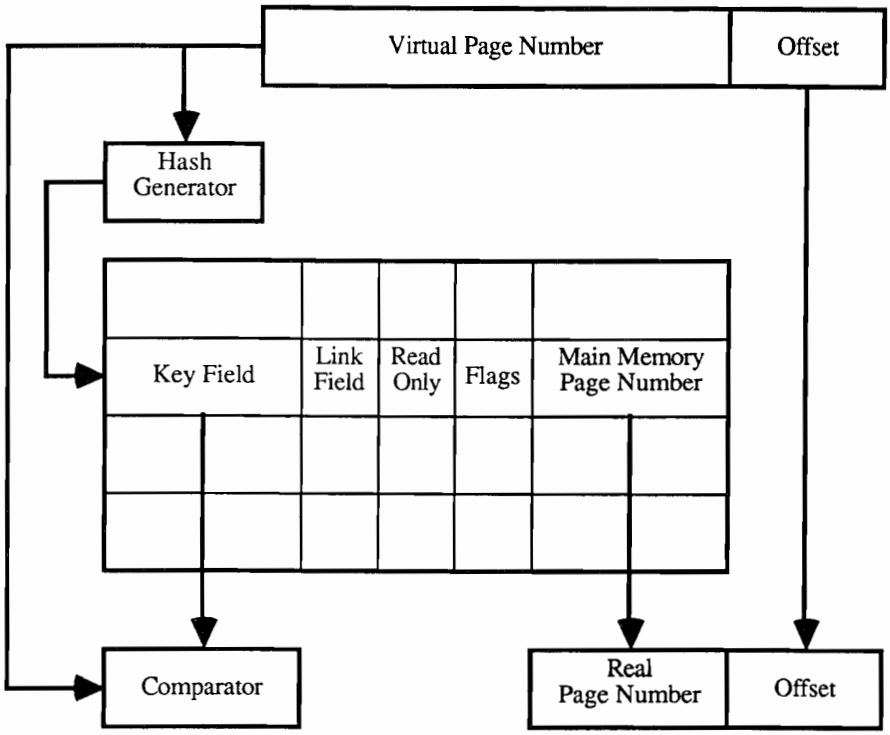

Figure 3: The Monads address translation hardware.

particularly complex and are implemented in microcode or in the kernel of the operating system. Because addresses in the Monads system have a system-wide validity the contents of the hash table are not affected by context switches. The MU6-G appended the process number in order to ensure uniqueness of virtual addresses across processes and achieve the same effect.

Given that the address translator must be able to translate all virtual addresses of pages in main memory, the number of entries in the table must be at least as large as the number of page frames. In order to keep the number of synonyms small the table must be considerably larger. Assuming a random uniform distribution of cell addresses it can be shown that the average search length is given by the formula:

$$1 + (\alpha/2)$$

where α is the loading factor, i.e. the ratio of full cells to total size [42]. For example, with a loading factor of 0.25 (corresponding to a table with a number of entries equivalent to four times the number of pages of main memory), the average search length is 1.125. A significant feature of this scheme is that the length of the table (and therefore the cost of an implementation) increases linearly with the size of *main memory*. Furthermore the table width increases only *logarithmically* with the size of the virtual memory. Therefore, increasing from a conventional virtual address size of 32 bits to a very large virtual memory with 128 bit addresses does not significantly increase the cost of address translation.

The above discussion has assumed a random uniform distribution of cell addresses. It is unlikely that virtual addresses will be uniformly distributed and therefore they must be hashed to form a cell address. The hashing function is performed on every address translation and thus must be simple. Such a simple function may be implemented by performing an exclusive OR of selected bits.

If the virtual address is found in the head of a synonym chain then the address translation time is the same as on a conventional lookaside buffer. The fetching of entries in a chain may be overlapped with the key comparison, minimising the overheads of following a chain. The address translation time is relatively insensitive to the size of both virtual and main memory addresses and to lack of locality of reference.

The Monads-PC implementation supports 60 bit virtual addresses

which are translated into 24 bit main memory addresses. The major advantage of this scheme is that the tables required to manage secondary storage may be completely independent of the hardware, leaving considerable flexibility. Alternative address mapping techniques with similar properties are described in [49, 59].

The Rekursiv [32] developed by Linn Smart Computing uses a different approach to supporting a large address space. Instead of mapping objects onto a paged address space, an object space is supported. Each object has a unique identity of 39 bits. The top bit of these object identifiers partitions the object space into compact types and ordinary types. Compact types are self-referential with the object value and identity being the same. They are used to represent scalars in an efficient manner. All objects have a unique identity and are referenced by specifying the identity and an index. The system maps the object id onto memory addresses and automatically copies objects to and from memory and backing store on demand. This is effectively a segmented virtual memory.

Hardware assistance is provided for the address translation process which is confusingly called "paging." The address translation hardware is shown in Figure 4. As in the Monads system the "pager" table is held in dedicated high-speed memory. On a memory access the lower 16 bits of the object id are used as an index into the table. The remaining bits of the id are compared with the object id in the table. If there is a match then the object is currently in memory otherwise a fault occurs. On a fault the software must locate the object on secondary storage, copy it into free memory and insert an entry in the "pager" table.

If the object id matches, then the index is compared with the size to verify that the reference is within the bounds of the object. Optionally (under microcode control) the type may be checked to ensure that only appropriate operations are performed on the object. The index is then added to the memory address of the object to generate a final physical address for the access. These steps are actually overlapped as far as possible. The status bits are used by the discard algorithm. As an optimisation the first word of each object is actually held in the "pager" table (object representation) so that single word objects can be handled efficiently.

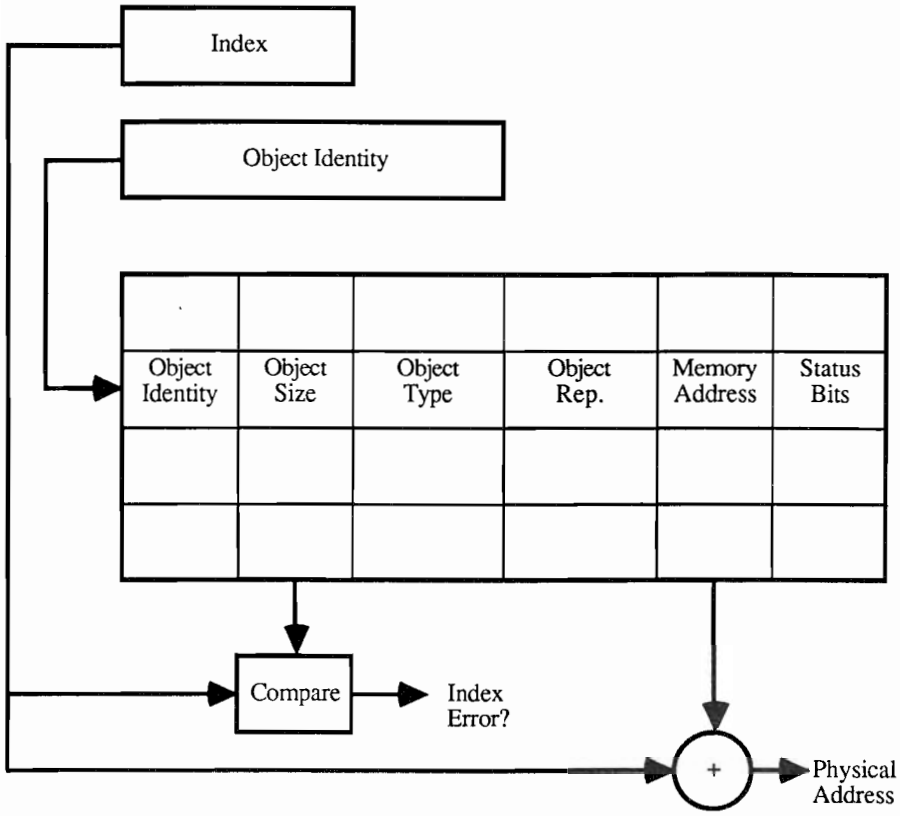

Figure 4: The Rekursiv address translation hardware.

Note that the hardware does not handle clashes in the "pager" table. That is if two objects have the same lower 16 bits in their object id then they cannot both be in the table at the same time. This effectively means that two objects which match on the lower 16 bits cannot be in memory at the same time. It is argued by the designers that this is unlikely and can be partially avoided by careful allocation of object ids, but this argument is not very convincing. In addition since objects rather than pages are swapped between main and secondary storage the system designer is left with the difficult problem of managing and locating variable length objects on secondary storage. Nevertheless, the Rekursiv has been manufactured, including three custom designed

VLSI chips, and an operational system has been produced as a co-processor to a SUN server system.

A final area of potential for architectural support of memory management for persistent object systems is garbage collection. The nature of persistent systems is such that garbage collection is essential and can be quite costly in terms of performance. A large amount of work is being done in this area and a number of schemes which utilise virtual memory hardware have been proposed (e.g. Kolodner [38]). Most of these schemes only require the ability to control protection of pages and detect violations of this protection. However, this is an area which will take on increasing importance as the size of stores grows.

3.2 Protection

A major issue in the design of persistent object systems is protection. Given that all objects reside within a single large persistent store it is essential to have some control over which objects may be accessed and to ensure the type security of those objects. In a conventional system long-term data is essentially grouped into files and the file system provides mechanisms to support controlled access to these files. In a persistent system the store consists of a graph of objects and protection is achieved by limiting the ability of a process to create a reference to an object. In fact, most persistent systems only allow a reference to an object to be obtained by creating a new object or by extracting a reference from an existing object; references may not be arbitrarily manufactured. Thus protection may be achieved by restricting the ability of a process to traverse the graph of objects, i.e. to follow pointers. We will show how this may be implemented later in this paper. Type security is usually enforced by the compilation and run-time system of a programming language.

There are two basic approaches to the provision of protection. In many persistent systems a single language is employed and all applications live within this language environment [24]. In such systems the type security is sufficient to build arbitrarily complex protection systems via information-hiding [45]. However, these systems are overly restrictive in the sense that the security is dependent on the use of a single language.

The alternative is to provide some hardware support for protection. Existing hardware provides some assistance in that most machines support multiple address spaces. In theory it would be possible to place each object in a separate address space and then provide an access mechanism for controlling which objects may be addressed. The difficulty is that the minimum size of an address space is usually one page and in most cases only one address space is accessible at any instant. These address spaces are intended to be used for independent processes and do not map well onto the persistent object paradigm.

Most of the proposed hardware-based protection mechanisms have been based on the use of capabilities. Capabilities were first proposed by Dennis and Van Horn [27] as a technique for describing the semantics of controlled access to data. The idea was extended by Fabry who proposed a computer system based on capabilities [29]. There have been several attempts at constructing such a capability-based system. Some of these enlisted hardware support [46, 52, 60] and others were purely software implementations [62]. Although these systems differ greatly the fundamental principles of capability-based addressing are the same.

The basic idea is that access to objects is controlled by the ownership and presentation of capabilities. That is, in order for a program to access an object it must produce a capability for the object. In this sense capabilities may be viewed as keys which unlock the object to which they refer. Since the possession of a capability gives an undeniable right to access an object it is important that programs not be able to manufacture capabilities. Such an ability would allow a program to access data which was not supposed to be available to it. Methods of protecting capabilities include segregation [47, 62], tagging [30, 46] and password schemes [11]. A capability for an object can thus only be obtained by creating a new object or by being passed a capability by another program.

Capabilities have three components. These are a unique name identifying the object, a set of access rights for the object identified by the capability and access rights for the capability itself. Capability systems use object names which are unique for the life of the system. The name given to an object will never be re-used, even if the object is deleted. This avoids aliasing problems and provides a means of trap-

ping dangling references. Such unique names are not difficult to generate and identifiers in the order of 64 bits are sufficient to ensure that the system will never exhaust all possible names.

Although the ownership of a capability guarantees the right to access the corresponding object, the object access rights field may restrict the level of access allowed. The facilities provided by access rights vary greatly between different capability systems. They may be as simple as read, write and execute, or they may be based on the semantics of the different objects, for example a list of procedures for accessing an abstract data type. When a capability is presented in an attempt to access an object the system checks that the type of access required does not exceed that allowed in the capability. There is usually an operation which allows a new capability to be created from an existing one with a subset of the access rights. This allows for the construction of restricted views.

The third field of a capability contains access rights which indicate which operations can be performed on the capability itself. Again, these vary greatly. The minimum usually provided is a "no copy" right which restricts the copying of the capability, perhaps on a user basis. This may be used to stop a user from passing a capability on to other users, i.e. to limit propagation. Other access rights may include a "delete" right which allows the holder of the capability to delete the object.

A final facility provided on some capability systems is the ability to revoke access. That is, after giving a program a capability it may be desirable at a later time to revoke this capability. Implementation of revocation is not easy. The simplest technique is to change the unique name of the object. This will effectively invalidate all existing capabilities. Selective revocation may be supported by using indirection through an owner controlled table of access rights or by providing multiple names for the object which can be individually invalidated.

Capabilities provide a uniform model for controlling access to data which synergises well with our uniform model for storage. A number of machines with hardware support for capability-based addressing of persistent objects have been constructed.

The Rekursiv described earlier effectively supports capability-based addressing. Recall that each object has a unique object id and an ob-

ject can only be accessed by providing its id. Object store words on the Rekursiv are 40 bits in size. The top bit of each word is a tag which indicates if the corresponding word contains an id (i.e. is a capability) or data. The tags cannot be modified by user programs thus providing the required level of protection for capabilities. The Rekursiv also associates a size field with capabilities so that the bounds can be checked on index operations.

The Rekursiv scheme has a number of drawbacks. First, the use of tags creates a number of difficulties including the size of memory, management on secondary storage devices and the additional overhead of checking the tags. These problems have been well described in the literature [31]. Second, since all objects are managed by the one capability scheme the same level of overhead is incurred for all object accesses (although there is an optimisation for scalar access). For example an entry in the "pager" table will be used for every structure. Third, the number of bits (39) is insufficient to guarantee uniqueness and eventually object ids have to be re-used. This creates a massive garbage collection problem.

The Monads system also supports capability-based addressing of objects, however a distinction is made between large objects, called *modules,* and small objects, called *segments*. Modules are used to represent major software resources such as programs, files and operating system entities. They consist of encapsulated data and procedures for accessing that data. Modules are protected by *module capabilities* which define the access allowed to the module in terms of semantic operations appropriate to that object. Segments are used to represent small objects such as integers and records. They are protected by a lower level mechanism based on *segment capabilities,* which still guarantees security and protection. Dasgupta [23] makes a similar distinction.

The reason for the distinction is that the frequency and style of access varies between large and small objects and this separation allows an efficient implementation for small objects without compromising flexibility for large objects. The architecture does not enforce any minimum size rules and any object may be modelled as a module or segment.

Both module and segment capabilities are protected by the archi-

tecture so that it is not possible for a program to manufacture or modify a capability. A capability is provided by the system whenever a new object is created. The only other way to gain access to an object is to be given a capability by another program. A module's internal data, called its database, consists of segments containing data which may include module and segment capabilities.

All data in the Monads system is held in arbitrary sized segments, each of which has the same basic format, with three sections as illustrated in Figure 5. The control section defines the size of the segment and details about the contents of the information section. Segments may contain module capabilities, one of a small number of system defined types, or arbitrary data. The architecture restricts access to the information section appropriately for the type it contains. For example, arbitrary modifications may be performed on ordinary data, whereas module capabilities may only be assigned and used to call another module. The capability section contains references to other segments so that arbitrarily complex graph structures of segments may be produced. The segments are mapped onto the paged virtual store described earlier, and may overlap page boundaries in an arbitrary fashion.

Segments may only be addressed by segment capabilities. A seg-

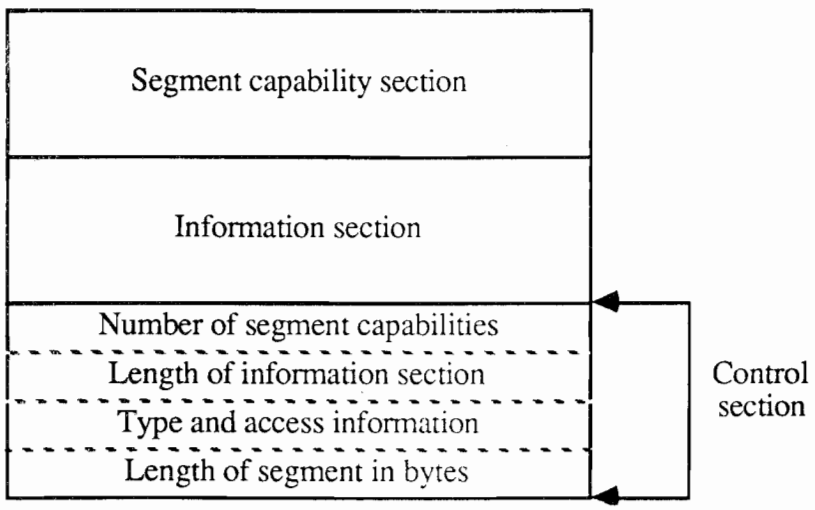

Figure 5: Segment format.

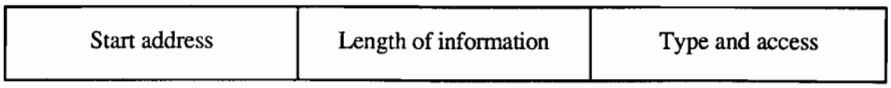

Figure 6: Capability register format.

ment capability need only contain the starting virtual address of the identified segment since segments are self describing. For efficiency, the architecture provides a set of addressing registers, called *capability registers,* of the format shown in Figure 6.

In order to address a segment the segment capability is loaded into a capability register, with the information length, type and access being obtained from the segment itself. Machine instructions therefore address data using addresses of the form <capability register number> <offset within segment>. The architecture guarantees that any attempt to access data outside of the information section will cause a run-time exception. This provides implicit support for array bounds checking. This addressing is no more expensive than addressing on a conventional base register machine since the length and access check can be overlapped with the address computation.

Loading of capability registers is obviously critical to system security. A special machine instruction is provided for this purpose. This instruction will only load a capability register with a segment capability contained in a segment currently pointed to by another capability register. Using this facility it is possible to traverse arbitrary data structures in a secure fashion. Similarly a machine instruction is provided to store segment capabilities in the segment capability section of a segment. This instruction enforces certain rules which limit the propagation of segment capabilities in the system. These rules simplify the problem of garbage collection and allow most segment capabilities to be maintained in an abbreviated form. The rules are described elsewhere [53].

The root of all addressing is defined by a special segment, of which there is one per process, called the *base table*. An additional machine instruction is provided to load a capability register to point at the base table. By changing the contents of the base table the current addressing environment may be modified. This is the key to the module protection scheme.

The code of procedures is also held in segments. These are usually not directly addressable from the base table. Rather there is a dedicated capability register which holds a segment capability for the current procedure. This is automatically changed on a procedure call. The architecture ensures that a procedure can only exit via a return instruction. A jump to outside of the code of the current procedure will cause an exception.

When a module is compiled, the compiler produces the procedure segments and two additional segments. The first, called the *interface entry-point list*, contains a segment capability for each interface procedure of the module (i.e. procedures which can be called by other modules). The second, called the *internal entry-point list*, contains an equivalent set of segment capabilities for procedures internal to the module. The locations of these entry-point lists for the current module are always available to the architecture via some red-tape information associated with each module. There are two procedure call instructions, one for internal calls and one for inter-module calls. The call instructions specify the procedure to be called as an index into the appropriate entry-point list. Thus the call instruction may retrieve the segment capability for the called procedure.

On each procedure call a new segment is created. Parameters are passed by either copying the value into this segment (pass by value) or by copying a segment capability into the segment (pass by reference). The call mechanism adds a segment capability for this new segment into the base table to make it addressable in the called routine and removes it on the return. A similar mechanism is used to create and delete local data for the called routine.

In order to call a procedure of another module the program must present a valid module capability. These have the format shown in Figure 7. The module name provides sufficient information to the architecture to allow the entry-point lists and a root segment for the private data of the module to be located. The mechanism used for this purpose is described in [53]. The access rights define which of the

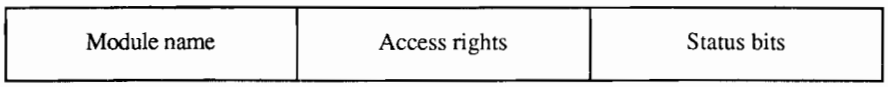

Figure 7: A module capability.

procedures of the identified module may be called using this module capability. The status bits define which operations may be performed on the capability itself, e.g. whether it can be copied.

The inter-module call instruction checks the supplied module capability to ensure that the procedure to be called is included within the access rights. It then transfers control to the procedure, obtaining a segment capability for the procedure from the interface entry-point list. The call mechanism modifies the base table to include a capability for the root segment of the private data for the called module and to remove and save the segment capability for the root segment of the private data of the calling module. That is, the call mechanism causes a change of protection domain. Currently the call mechanism provides no support for inheritance but it would not be difficult to add this facility using, for example, the technique described by Connor et al [21].

The major advantage of this two level scheme is that direct access to scalars and small data structures is achieved at a speed comparable to conventional architectures using the segment level of addressing, while the power and flexibility of module capabilities can be used for information-hiding objects.

3.3 Distribution

Given the current proliferation of workstations with local storage connected via a network, distribution of the store has become an important issue. In conventional systems distribution is usually achieved via the file system. Systems such as NFS [58] provide more or less transparent access to files on remote machines. Persistent systems have no notion of files and thus a different approach must be used.

One approach is to extend the persistent store to encompass the entire network. This involves supporting addresses which are large enough to address all data on all nodes in the network. The Monads system [35] and the distributed Napier88 system [36] both employ this technique and both are based on supporting a large distributed shared memory [39]. Napier88 utilises a central server approach whereas Monads supports multiple servers.

In the Monads system any machine on the network with an appropriate capability may generate an address of any object on any machine in the network. If the page containing the object is not resident on the requesting node then a message requesting a copy of the page is

sent to the owner node. Coherence is guaranteed at the page level using a multiple-reader single-writer protocol. This is also the case with the Napier88 system. The Monads system supports full naming and location transparency and migration of objects between nodes via a system of forwarding addresses and advisory information. This scheme is described in [33].

The distributed shared memory approach requires little additional architectural support, assuming that addresses are large enough. Effectively all that is required is the ability to control the read-write protection on a page basis so that the first modification to a page can be detected. This is required in order to efficiently implement the multiple-reader single-writer protocol.

A major problem with the distributed shared memory approach to distribution is reliability. At any instant in time there may be pages from any number of nodes held within the memories of any number of other nodes. If a particular node should crash then an arbitrary set of pages and modifications will be lost. Various proposals for distributed stable stores have been made [34, 36]. These usually rely on extending the idea of shadow paging [41] to include pages distributed throughout the network. The difficulty is that the checkpoint operation, which saves the current state of the store, requires retrieval of all modified pages and this may not be possible if a node is down. There appears to be no satisfactory scalable solution to this problem.

The problem of reliability in distributed persistent systems is discussed in [26]. An alternative approach based on message passing and exportation of code to be executed on a remote machine is described. In this scheme the persistent store of each machine is essentially independent and direct references between stores are prohibited. This has the advantage that hardware-supported addresses need only be large enough to access the local store and it also allows checkpointing of an individual store to take place at any time. However, it offers less flexibility than the distributed shared memory system since applications must be aware of the distribution and concurrency.

4. A Persistent RISC

In this section we briefly describe a new machine being constructed by the Persistent Systems Group at the University of Sydney. The machine has been specifically designed to support the requirements of

persistent object systems. Only a brief description is given here since the project has been more fully described elsewhere [37, 54].

The new machine is broadly based on the Monads architecture described above. However, rather than develop our own central processor as with the Monads-PC, we have chosen to utilise a SPARC [57] processor unit. This provides us with a level of compatibility with existing workstations and an upgrade path as faster versions of the processor become available.

The SPARC architecture only supports 32 bit addresses and this creates a major difficulty. The approach taken has been to add additional addressing hardware external to the processor. This hardware consists of a set of addressing registers similar to the capability registers described for the Monads-PC machine. These mirror the integer sliding register windows and are managed in a similar fashion. Each capability register contains the starting virtual address of the corresponding segment (a 128 bit address) as well as the length and access rights. On each memory access a capability register and an offset relative to this register are specified. The external addressing hardware adds the offset to the base address from the capability register and sends the resultant virtual address to an address translation unit based on the hashing scheme described in section 3.1. The checking of the length field and access rights takes place in parallel with the access.

The difficulty with this scheme is that it requires both a capability register number (5 bits in order to address the 32 capability registers) and an offset (ideally 32 bits) to be specified on each memory access. This is difficult since the SPARC only emits a 32 bit address. Two solutions were considered. The first involves using the top 5 bits of the address to identify the capability register and the remaining 27 bits as the offset. A similar scheme has been used on other machines [8, 48]. This scheme has several disadvantages. First, it reduces the offset size to 27 bits and thus the maximum size of a segment to 2^{27} bytes. Second, it is most convenient if the offset is in the low order bits of the address and this means that the register number is in the high order bits. As a result almost all addresses have significant bits (which are known at compile time) in the high order bytes. Such large constants are difficult to construct on the SPARC. Third, one of the advantages of capability-based addressing is that there is automatic bounds checking. If the high order bits are used as a register number then this bounds checking is lost, since an index error may result in the register

number field of an address being inadvertently modified. As a result the generated address may still be valid but may not be pointing to the intended data structure.

For all of these reasons we have rejected this solution and have adopted a more pragmatic approach to the problem. Ideally we would re-design the SPARC to provide large addresses. This option is not available to us and so we simulate it. For every load and store instruction the compiler generates an additional load instruction preceding the actual memory reference. The first load instruction has as its address the capability register number. This load is "consumed" by the addressing hardware and completes in minimum time. The actual memory reference has as its address the offset, which can be up to 32 bits. The capability register can be fetched during the first load so that the addition can take place as soon as the offset is available. An advantage of this approach is that the existing SPARC addressing modes can be used to generate the offset.

There are two costs associated with the scheme. The first is an increase in code size of approximately 25 percent, assuming that about 25 percent of instructions are loads or stores. Since the machine has a substantial instruction cache it is not expected that this will have a serious impact on performance. The second cost is the overhead of the execution of an extra load instruction for each load and store. In the worst case this will increase execution time by 25 percent, although it may well be less since many memory references take longer than the minimum time. We are willing to accept this cost in order to maintain the integrity of the architecture.

The call mechanism and instructions for loading the capability registers are provided as system subroutines. Since code for these is held in dedicated high speed memory on the processor board, these subroutines may execute at the maximum clock speed of the SPARC processor. The kernel is also held in dedicated memory and does not need to use the extra loads since it is held in a well-known region of the virtual address space.

An additional feature of the machine is support for a very large main memory of up to 64 gigabytes. This is managed by having two page sizes as described in [54]. The intention is to perform experiments in the use of massive memory to achieve high-speed execution of certain algorithms and applications, and for an investigation of

large main memory databases. The persistence paradigm lends itself easily to this type of experimentation.

At the time of preparation of this paper the designs of the memory modules and address translation hardware are complete and they are currently being constructed. Design of the processor module is underway and it is hoped that a prototype will be operational soon.

5. Conclusion

Persistent object systems have the potential to provide significant improvements in productivity, both in terms of software development time and ease of use of the resulting systems. This technology will not be acceptable until it achieves at least comparable performance to conventional systems. It is unreasonable to expect conventional architectures, which were essentially designed to support large monolithic programs and a file store, to efficiently support scalable persistent systems. We should therefore be investigating the requirements of these systems so that they can be incorporated in the next generation of architectures.

Two major requirements have been identified. First, larger address spaces are required in order to provide direct addressing for large persistent stores. It is envisaged that address sizes in the order of 96 to 128 bits may be required. Alternative address translation mechanisms will be required to support these large addresses and to handle the possibly reduced locality of reference in persistent systems. Second, flexible protection mechanisms are required to provide for controlled access to data in the store.

The SPARC-based machine described in section 4 is a first attempt to provide such an architecture using modern RISC technology. It will provide a test bed for evaluating these requirements using existing persistent systems.

Acknowledgements

This work was partly supported by Australian Research Council grant A49031987. The author also wishes to thank Fred Brown and Frans Henskens for their improvements to several earlier drafts of this paper.

References

[1] "Proceedings of the International Workshop on Database Programming Languages," Roskoff, France, 1987.

[2] "Datatypes and Persistence," *Proceedings of Data Types and Persistence Workshop Aug. 1985,* Appin, Scotland, (ed M. P. Atkinson, P. Buneman and R. Morrison), Springer-Verlag, 1988.

[3] "PS-algol Reference Manual - fourth edition," University of Glasgow and
St Andrews, Persistent Programming Research Report 12/88, 1988.

[4] "Persistent Object Systems," *Proceedings of the 3rd International Workshop on Persistent Object Systems,* Newcastle, Australia, (ed J. Rosenberg and D. M. Koch), Springer-Verlag, 1989.

[5] "Proceedings of the International Workshop on Database Programming Languages," Salishan, U.S.A., Morgan Kaufmann, 1989.

[6] "Proceedings of the 4th International Conference on Persistent Object Systems," Martha's Vineyard, U.S.A., (ed A. Dearle, G. Shaw and S. Zdonik), Morgan-Kauffman, 1990.

[7] "Security and Persistence," *Proceedings of the International Workshop on Architectures to Support Security and Persistence of Information,* Bremen, Germany, (ed J. Rosenberg and J. L. Keedy), Springer-Verlag, 1990.

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

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

[10] Albano, A., Cardelli, L. and Orsini, R. "Galileo: A Strongly Typed, Interactive Conceptual Language," *ACM Transactions on Database Systems,* 10(2), pp. 230–260, 1985.

[11] Anderson, M., Pose, R. D. and Wallace, C. S. "A Password-Capability System," *The Computer Journal,* 29(1), pp. 1–8, 1986.

[12] 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., pp. 360–365, 1983.

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

[14] Bertis, V., Truxal, C. D. and Ranweiler, J. G. "System/38 Addressing and Authorisation," *I.B.M. System/38 Technical Developments,* pp. 51–54, 1978.

[15] Brown, A. L. "Persistent Object Stores," Universities of St. Andrews and Glasgow, Persistent Programming Report 71, 1989.

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

[17] Brown, A. L. and Rosenberg, J. "Persistent Programming Systems: An Implementation Technique," *Proceedings of the 4th International Workshop on Persistent Object Systems,* Martha's Vineyard, U.S.A., Morgan-Kaufmann, 1990.

[18] Chorus Systemes "Overview of the CHORUS© Distributed Operating Systems," CS/TR-90-25.1, 1991.

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

[20] Cockshott, W. P. and Foulk, P. W. "Implementing 128 Bit Persistent Addresses on 80x86 Processors," *Proceedings of the International Workshop on Computer Architectures to Support Security and Persistence of Information,* Bremen, West Germany, (ed J. Rosenberg and J. L. Keedy), Springer-Verlag and British Computer Society, pp. 123–136, 1990.

[21] Connor, R. C. H., Dearle, A., Morrison, R. and Brown, A. L. "An Object Addressing Mechanism for Statically Typed Languages with Multiple Inheritance," Proc. *Object-Oriented Programming: Systems, Languages and Applications,* New Orleans, Louisiana, pp. 279–286, 1989.

[22] Connor, R. C. H. and Morrison, R. "Subtyping Without Tears," *15th Australian Computer Science Conference,* Hobart, Australia, pp. 209–225, 1992.

[23] Dasgupta, P., LeBlanc, R. J. and Appelbe, W. F. "The Clouds Distributed Operating System," *Proceedings, 8th International Conference on Distributed Computing Systems,* 1988.

[24] Dearle, A. "On the Construction of Persistent Programming Environments," Ph.D. Thesis, University of St. Andrews, 1988.

[25] Dearle, A., Connor, R. C. H., Brown, A. L. and Morrison, R. "Napier88 - A Database Programming Language?," *Proceedings Second International Workshop on Database Programming Languages,* Portland, Oregon, Morgan Kaufmann, pp. 179–195, 1989.

[26] Dearle, A., Rosenberg, J. and Vaughan, F. "A Remote Execution Mechanism for Distributed Homogeneous Stable Stores," *Proc. 3rd International Workshop on Database Programming Languages,* Greece, 1991.

[27] Dennis, J. B. and Van Horn, E. C. "Programming Semantics for Multiprogrammed Computations," *Communications of the A.C.M.,* 9(3), pp. 143–145, 1966.

[28] 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," *Computer Architecture News,* 8(3), pp. 161–167, 1980.

[29] Fabry, R. S. "Capability-Based Addressing," *Communications of the A.C.M.,* 17(7), pp. 403–412, 1974.

[30] Feustal, E. A. "On the Advantages of Tagged Architectures," *IEEE Transactions on Computers,* C-22, 7, pp. 644–656, 1973.

[31] Gehringer, E. F. and Keedy, J. L. "Tagged Architecture: How Compelling are its Advantages?," *Proc. 12th International Symposium on Computer Architecture,* Boston, Mass., pp. 162–170, 1985.

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

[33] Henskens, F. A. "A Capability-Based Persistent Distributed Shared Memory," Ph.D. Thesis, University of Newcastle, 1991.

[34] Henskens, F. A., Rosenberg, J. and Hannaford, M. R. "Stability in a Network of MONADS-PC Computers," *Proceedings of the International Workshop on Computer Architectures to Support Security and Persistence of Information,* Bremen, West Germany, (ed J. Rosenberg and J. L. Keedy), Springer-Verlag and British Computer Society, pp. 246–256, 1990.

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

[36] Koch, B., Schunke, T., Dearle, A., Vaughan, F., Marlin, C., Fazakerley, R. and Barter, C. "Cache Coherence and Storage Management in

a Persistent Object System," Proceedings, *The Fourth International Workshop on Persistent Object Systems,* Martha's Vineyard, Massachusetts, U.S.A., pp. 99–109, 1990.

[37] Koch, D. M. and Rosenberg, J. "A Secure RISC-Based Architecture Supporting Data Persistence," *Proceedings of the International Workshop on Computer Architectures to Support Security and Persistence of Information,* Bremen, West Germany, (ed J. Rosenberg and J. L. Keedy), Springer-Verlag and British Computer Society, pp. 188–201, 1990.

[38] Kolodner, E. "Atomic Incremental Garbage Collection and Recovery for a Large Stable Heap," *Implementing Persistent Object Bases, Proceedings of the Fourth International Workshop on Persistent Object Systems,* Martha's Vineyard, (ed A. Dearle, G. M. Shaw and S. B. Zdonik), Morgan Kaufmann, pp. 185–198, 1990.

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

[40] Liskov, B. "The Argus Language and System," *Lecture Notes in Computer Science,* vol 190, Springer-Verlag, New York, 1985.

[41] Lorie, R. A. "Physical Integrity in a Large Segmented Database," *ACM Transactions on Database Systems,* 2,1, pp. 91–104, 1977.

[42] Morris, R. "Scatter Storage Techniques," *Communications of the ACM,* pp. 38-43, 1968.

[43] Morrison, R. and Atkinson, M. P. "Persistent Languages and Architectures," *Proceedings of the International Workshop on Computer Architectures to Support Security and Persistence of Information,* Bremen, West Germany, (ed J. Rosenberg and J. L. Keedy), Springer-Verlag and the British Computer Society, pp. 9–28, 1990.

[44] Morrison, R., Brown, A. L., Conner, R. C. H. and Dearle, A. "Napier88 Reference Manual," Universities of Glasgow and St. Andrews, Persistent Programming Research Report PPRR-77-89, 1989.

[45] Morrison, R., Brown, A. L., Connor, R. C. H., Cutts, Q. I., Kirby, G., Dearle, A., Rosenberg, J. and Stemple, D. "Protection in Persistent Object Systems," *Proceedings of the International Workshop on Computer Architectures to Support Security and Persistence of Information,* Bremen, West Germany, (ed J. Rosenberg and J. L. Keedy), Springer-Verlag and British Computer Society, pp. 48–66, 1990.

[46] Myers, G. J. and Buckingham, B. R. S. "A Hardware Implementation of Capability-Based Addressing," *Operating System Review,* 14, 4, 1980.

[47] Needham, R. M. and Herbert, A. J. "The Cambridge Distributed Computing System," Addison Wesley, London, 1982.

[48] Pose, R. D. "Capability Based, Tightly Coupled Multiprocessor Hardware to Support a Persistent Global Virtual Memory," *Proceedings of the 22nd Annual Hawaii International Conference on System Sciences,* Hawaii, U.S.A., (ed B. D. Shriver), pp. 36–45, 1989.

[49] Ramamohanarao, K. and Sacks–Davis, R. "Hardware Address Translation for Machines with a Large Virtual Memory," *Information Processing Letters,* 13(1), pp. 23–29, 1981.

[50] Richardson, J. E. and Carey, M. J. "Implementing Persistence in E," *Proceedings of the Third International Workshop on Persistent Object Systems,* Newcastle, Australia, (ed J. Rosenberg and D. M. Koch), Springer–Verlag, pp. 175–199, 1989.

[51] Rosenberg, J. "The MONADS Architecture - A Layered View," *Proceedings of the 4th International Workshop on Persistent Object Systems,* Martha's Vineyard, U.S.A., Morgan–Kaufmann, 1990.

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

[53] Rosenberg, J. and Keedy, J. L. "Object Management and Addressing in the MONADS Architecture," *Proceedings of the International Workshop on Persistent Object Systems,* Appin, Scotland, 1987.

[54] Rosenberg, J., Koch, D. M. and Keedy, J. L. "A Massive Memory Supercomputer," *Proc. 22nd Hawaii International Conference on System Sciences,* vol 1, pp. 338–345, 1989.

[55] Schmidt, J. W. "Some High Level Language Constructs for Data of Type Relation," *ACM Transactions on Database Systems,* 2(3), pp. 247–261, 1977.

[56] Stroustrup, B. "The C++ Programming Language," Addison-Wesley, 0-201-12078-x, 1986.

[57] Sun Microsystems Inc. "The SPARC Architecture Manual, Version 7," Part No: 800-1399-08, 1987.

[58] Sun Microsystems Inc. "Systems and Networks Administration," Part No: 800-1733-10, Revision A, 1988.

[59] Thakker, S. S. and Knowles, A. E. "Virtual Address Translation Using Parallel Hashing Hardware," *Proceedings of the Supercomputing Systems Conference,* Florida, I.E.E.E., pp. 697–705, 1985.

[60] Wilkes, M. V. and Needham, R. M. "The Cambridge CAP Computer and its Operating System," Elsevier North Holland Inc., 1979.

[61] Wilson, P. R. "Pointer Swizzling at page Fault Time: Efficiently Supporting Huge Address Spaces on Standard Hardware," *Computer Architecture News,* vol 19, 4, pp. 6–13, 1991.

[62] Wulf, W. A., Levin, R. and Harbison, S. P. "HYDRA/C.mmp: An Experimental Computer System," McGraw–Hill, New York, 1981.

[received March 23, 1992; accepted May 3, 1992]