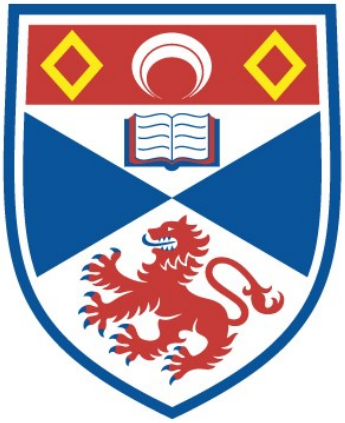
PERSISTENT OBJECT STORES

Alfred Leonard Brown

A Thesis Submitted for the Degree of PhD
at the
University of St Andrews



1989

Persistent Object Stores

Alfred Leonard Brown

Department of Computational Science

University of St. Andrews

October 1988

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 – 1346

Declarations

I Alfred Leonard Brown hereby certify that this thesis has been composed by myself, that it is a record of my own work, and that it has not been accepted in partial or complete fulfilment of any other degree or professional qualification.

Signed: Date: 28/10/88

I was admitted to the Faculty of Science of the University of St. Andrews under Ordinance General No. 12 on 1st October 1983 and as a candidate for the degree of Ph.D. on 1st February 1985.

Signed: Date: 28/10/88

I hereby certify that the candidate has fulfilled the conditions of the Resolution and Regulations appropriate for the Degree of Ph.D.

Signature of Supervisor: Date: 28/10/88

In submitting this thesis to the University of St. Andrews I understand that I am giving permission for it to be made available for use in accordance with the regulations of the University Library for the time being in force, subject to any copyright vested in the work not being affected thereby. I also understand that the title and abstract will be published, and that a copy of the work may be made and supplied to any bona fide library or research worker.

Abstract

The design and development of a type secure persistent object store is presented as part of an architecture to support experiments in concurrency, transactions and distribution.

The persistence abstraction hides the physical properties of data from the programs that manipulate it. Consequently, a persistent object store is required to be of unbounded size, infinitely fast and totally reliable. A range of architectural mechanisms that can be used to simulate these three features is presented. Based on a suitable selection of these mechanisms, two persistent object stores are presented.

The first store is designed for use with the programming language PS-algol. Its design is evolved to yield a more flexible layered architecture.

The layered architecture is designed to provide each distinct architectural mechanism as a separate architectural layer conforming to a specified interface. The motivation for this design is two-fold. Firstly, the particular choice of layers greatly simplifies the resulting implementation and secondly, the layered design can support experimental architecture implementations. Since each layer conforms to a specified interface, it is possible to experiment with the implementation of an individual layer without affecting the implementation of the remaining architectural layers. Thus, the layered architecture is a convenient vehicle for experimenting with the implementation of persistent object stores.

An implementation of the layered architecture is presented together with an example of how it may be used to support a distributed system. Finally, the architecture's ability to support a variety of storage configurations is presented.

Acknowledgements

I would like to thank my supervisor, Professor Ron Morrison for his guidance and support in the preparation of this thesis. I would also like to thank the following, Al Dearle for forcing the pursuit of 'niceness' even when we were not sure what 'nice' was, Ray Carrick and Richard Connor for testing the variety of systems that emerged, Chris Marlin for the red ink, Pete Bailey and Paul Cockshott for supplying some of the initial training and finally Professor Malcolm Atkinson for his very enthusiastic ideas.

Contents

Figures

1 What is persistence?

Persistence is defined to be the length of time for which data exists and is usable[atk83c]. It is an abstraction over one physical property of data, the length of time it is kept. In a persistent system, the use of all data is independent of its persistence. Thus, the use of the persistence abstraction removes the need to explicitly program for the differences in the use of long and short term data.

1.1 Why do we need persistence?

The persistence of data in a computer system can be described by the following six categories[atk83a]:

a) data that only exists within the evaluation of an expression,

b) data that is local to a procedure activation,

c) data that is global to a program or outlives the procedure that created it,

d) data that exists between executions of a program,

e) data that exists between versions of a program and

f) data that outlives the program that created it.

In traditional computer systems, categories **a** to **c** are usually supported by a programming language, whereas the remaining categories are usually supported by a file system or a database management system. This separation of support for data between programming languages and database or file systems introduces an explicit distinction between short and long term data. In such systems, any program wishing to use long term data must include explicit code for moving that data to and from a database or file system.

The distinction between long and short term data arose because of the relative size and cost of storage devices in the late 1960s and early 1970s. Main memory was fast but relatively small and expensive, whereas disk storage was slow but relatively large and cheap. This is

reflected in the different data structuring facilities available in programming languages and database systems. For example, a database system provides data structures specifically designed to manipulate large quantities of data held on disk. These data structures are necessary both to minimise the number of accesses made to the relatively slow disk storage device and also to allow the large quantities of data to be conveniently organised. In contrast, a programming language only operates on relatively small quantities of data within fast storage, so the data structures it provides are designed for expressive power. This discrepancy in the structuring facilities further aggravates the potential problems of using long term data within a program, since the program must also explicitly translate between data formats. This can be a complex operation, particularly if the programming language supports recursive data structures such as graphs, but the database or file system does not.

It has been estimated that as much as 30% of the code in a typical database application is concerned with the movement and organisation of the data and not in modelling the application[ibm78a]. A significant cost saving could therefore be achieved by integrating the use of long and short term data. To perform this integration, it is necessary to eliminate the distinction between programming languages and databases. This requires a mechanism that allows all data to be manipulated with the facilities of a programming language, while retaining the long term storage and bulk data facilities of a database or file system. This requirement is satisfied by the persistence abstraction.

1.2 Persistence

The implementation of the persistence abstraction presents its own problems, both intrinsic and technical. The intrinsic problems centre around the ideal facilities for persistent programming, such as:

a) how is persistent data identified,

b) are transactions necessary and

c) what naming facilities are provided?

The technical problems centre around the difficulties of efficiently engineering such a system, for example,

d) how is data managed in the hierarchy of storage media and

e) how is data protected from misuse?

1.2.1 How is persistent data identified?

The first of these problems is the identification of long term data, that is data that will persist after a program terminates. There are three major methods for identifying this persistent data:

a) all data persists,

b) only explicitly marked data persists and

c) all data reachable from one or more roots will persist.

The simplest method is to allow all data to persist. When a program terminates, all the data that it was using is saved as a single block of storage. When the data is required, the entire block of storage is retrieved. This method is known as *core dumping*.

Core dumping is used by Smalltalk[gol83] and some Lisp systems[tei78]. A major advantage of this method is that there are no restrictions on what data types can persist. Furthermore, an executing program could be made persistent if its dynamic state were included in the data to be saved. However, a disadvantage of this method is that the maximum amount of data that can be saved and retrieved is constrained by the maximum

memory size available to programs. Furthermore, to change one object, the entire persistent store must be loaded and then dumped.

The second method of identifying persistent data is to explicitly mark the data. That is, all movement of data between a program and the place it is stored is made explicit. An example of a system that uses explicit marking is Amber[car87]. Amber identifies persistent data by requiring it to be injected into a data type called *dynamic*. Once injected into the dynamic type, the data is explicitly saved in a named file using a command called *export*. Persistent data is explicitly retrieved from a named file using a command called *import*. Retrieved data is of type dynamic and must be projected onto its original type before it can be used. Furthermore, import makes a new copy of the retrieved data and cannot be used to obtain multiple references to the same data. The explicit marking of persistent data is unsuitable for persistent systems since the elimination of the explicit movement of data is one of the reasons for introducing the persistence abstraction.

The third method of identifying persistent data is by reachability from a set of specified roots. That is, any data reachable directly or indirectly from certain specified data objects is persistent. The construction of this transitive closure of persistent data requires that all data is organised into objects and that these objects describe all references they may contain to other objects.

An example of a system that uses reachability is PS-algol[psa88]. The PS-algol system organises its persistent store into separate units known as databases. The first object in each database is used as a root of persistence. Any data that is reachable from all of these root objects is automatically made persistent when a PS-algol program invokes the standard procedure *commit*. The retrieval of persistent data is automatically performed as the data is referenced. Thus, any particular persistent object can be retrieved by simply following object references from the appropriate root object.

The Argus system[lis84] is an example of how explicit marking can be combined with reachability. Argus identifies persistent data by allowing variables to be declared as stable. That is, persistent data can be explicitly declared by a programmer. Any data that is reachable from a stable variable is made persistent whenever a transaction commits. Hence, Argus also employs reachability to conveniently identify potentially complex data structures.

1.2.2 Are transactions necessary?

A transaction is an abstraction mechanism that allows a group of update operations to be treated as a single atomic action. That is, all the update operations succeed or none of them does. Transactions exist in database and file systems in various forms.

One of the uses of transactions in database and file systems is the saving of data in a consistent state on non-volatile storage. This second mechanism is essential for any system that contains valuable data. Furthermore, the first mechanism can be constructed from the second. Therefore, if the use of a database or file system is replaced by the persistence abstraction, then at least the second mechanism must be provided.

1.2.3 What naming facilities are provided?

There are a wide range of naming techniques that may be applied to a large volume of data. However, to be effective, the data must be divided into identifiable objects. These objects can vary in complexity from a relation to a string of characters. The possible naming techniques include centralised dictionaries, hierarchical directory structures, graphs of directory structures and programming language block structure and encapsulation.

1.2.4 How is data managed in the hierarchy of storage media?

The hierarchy of storage media available in a computer system includes tape archives, magnetic disks and main memory. Within this hierarchy, long term data is usually held on tapes or disks, whereas the short term data used by a programming language is held in main

memory. Since the persistence abstraction allows programming language facilities to be used on all data, it is desirable to treat all storage as main memory. This can be achieved by automating the movement of data between disks and memory to give the illusion that all the data is in main memory.

The core dumping technique can be automated very simply. When a program is started, all the data is loaded into memory and when it terminates all the data is written back to disk.

Other techniques for identifying persistent data require more sophisticated automation mechanisms. These mechanisms may operate at the physical level on the storage in which the data resides or at the logical level on individual data objects.

An automation mechanism operating at the physical level may be implemented by *paging*. That is, the storage holding the data is allocated a range of addresses in a program's address space. A *virtual addressing* mechanism is then used to move those disk blocks holding the data being used between memory and disk as and when required.

An automation mechanism working on individual data objects is very similar to a paging mechanism. This type of mechanism is known as *segmentation* or *object addressing*. It differs from paging in several significant ways, the most obvious, being that this mechanism moves objects instead of disk blocks between disk and memory. The movement of non-uniform sized objects introduces the classical problems of storage fragmentation. The major advantage of object addressing over paging is that the addressing of objects can be performed by a variety of different mechanisms.

1.2.5 How is data protected from misuse?

A major advantage of the persistence abstraction is that all the protection facilities of a programming language can be applied to long term data. In a database or file system, long

term data may be accessed by programs that are unaware of what the data represents. For example, if a program stores a floating point number in a file, another program could access the number as if it were a string of bytes. The byte string could then be transformed into a floating point number that could never have been generated by arithmetic. Such violations of a programming language's protection mechanisms are unacceptable in a type secure persistent store.

The protection provided by the persistence abstraction is dependent on the programming languages that may use it. This is because high level languages do not allow objects to be misused or misinterpreted, whereas low level languages may be unable to prevent this. Clearly the two types of languages require different protection mechanisms.

For a high level language, the protection of data may be performed by the language's type system. The strict enforcement of the type system ensures that a program cannot access a piece of data unless it knows the data's type. Furthermore, the program must interpret the data according to its type. An advantage of this protection mechanism is that most of the protection can be enforced in the writing of a program. In those cases where it cannot, a simple test at run time is usually sufficient to enforce the protection.

The protection mechanisms required for low level programming languages must be much more sophisticated since it is not possible to determine if a program may inadvertently access a piece of data with the wrong type or when it did not intend to. To protect data from such erroneous use, separate address spaces with access privileges are used. For example, in a UNIX system, every program is given at least two address spaces[rit74], a data space that can be read or written and a code space which can be read or executed. In addition to the two address spaces, the operating system has its own address spaces that user programs do not have the privilege to access.

A much finer grain use of address spaces is provided by a capability system[cos74,fab73]. A capability is the address of an object, plus a set of access privileges. In effect, each object is given its own address space. Within such a system, a program can only access an object if it has a capability for the object. In addition to protecting objects, a capability system also protects the capabilities. This is necessary to prevent erroneous programs modifying capabilities or manufacturing their own.

1.3 PS-algol and the CPOMS

The concept of persistence arose from work on integrating programming languages with database systems[atk78]. As part of this research, persistence was added to the programming language S-algol[mor82], requiring solutions to be found for each of the five problems described above. The resulting programming language, PS-algol[atk83c], was developed to test the hypothesis that persistence could be added to a programming language with a minimum of change. That is, as far as possible, the provision of persistence can be transparent. This has the following consequences:

a) the programmer does not have to explicitly identify persistent data,
b) all data may be used independently of whether or not it is persistent,
c) all data types may be made persistent and
d) since there is no distinction between persistent and non persistent data, the programming language's protection mechanisms apply to all data.

The development of PS-algol involved adding, to S-algol, a small set of interface procedures to access the persistent store and making procedures first class data types. S-algol provides a heap on which strings, vectors and structures are kept. It also allows all data types, except procedures, to be fields of a structure and thereby be held on the heap. This exception is removed in PS-algol by making procedures first class data types. Since the heap is a convenient abstraction over the storage of objects, the PS-algol persistent store is

modelled as a heap. Therefore, since all data types may be held on the PS-algol heap, they may also be made persistent.

The implementation of the PS-algol persistent store has been provided by a series of systems that manage persistent objects. These systems include the chunk management system[atk82], the PS-algol persistent object manager[atk83b] and, currently, the CPOMS (a persistent object management system written in C). The PS-algol/ CPOMS system adopted the following solutions to the five problems presented by persistence.

1.3.1 How is persistent data identified?

The PS-algol persistent store is organised into units called *databases*. The databases are used as roots of persistence from which persistent data can be identified. That is, the transitive closure of all data reachable from the first object in each database is persistent. Databases are also used as the units of data sharing.

1.3.2 Are transactions necessary?

PS-algol provides a transaction mechanism that allows programs to share databases. This mechanism is based on pessimistic concurrency control. That is, before a database can be used, it must be locked and any intention to modify it must be declared. The locking protocol applied to databases ensures exclusive access for updating but allows shared access for reading. When a program wishes to make changes to the persistent store permanent, it invokes a procedure called *commit*. This procedure first checks that all the databases to be updated were locked with the intention of being updated and then it makes the program's changes permanent. The commit is performed as an atomic action, so that either all the updates are made permanent or none of them are. This prevents the persistent store being left in an inconsistent state if a commit operation fails.

1.3.3 How is data managed in the hierarchy of storage media?

Data management in the PS-algol system is performed by an object addressing mechanism. The purpose of this mechanism is to copy objects from the persistent store to the heap of a PS-algol program. Within a heap, each copied object is given a local address with which it can be accessed quickly. The commit algorithm is used to copy objects from a program's heap to the persistent store.

1.3.4 How is data protected from misuse?

The protection provided by the PS-algol system is enforced by the PS-algol compiler and some simple run time tests. This involves the static type checking of all operations in the language with the addition of a run time type check when dereferencing a structure. The run time check is necessary since all PS-algol structures are accessed via an infinite union of all structure classes called *pntr*. Hence, all structure dereferences check that the structure being dereferenced is of the intended class.

The type checking of structure classes is based on structural equivalence. This allows structure classes compiled in different PS-algol programs to match the same structure class. Therefore, structures placed in the persistent store by one program can be used by another program, if the other program includes an equivalent structure class.

The protection can be enforced by the PS-algol compiler for two reasons:

a) PS-algol is a high level language. That is, programs cannot be written that misuse or misinterpret data objects.

b) PS-algol is the only language that can be run against the PS-algol system. Therefore, all programs are compiled by the PS-algol compiler and the objects they manipulate are subject to the PS-algol type system.

These properties of the PS-algol system allow the CPOMS to eliminate many of the protection systems required by persistent object managers for low level languages.

1.3.5 What naming facilities are provided?

The PS-algol system supports a naming mechanism for organising large quantities of data called a *table*. A table is a mapping from string and integer keys to the data type pntr. PS-algol allows all data types without exception to be fields of a structure. Therefore, any data type can be placed in a structure and then placed in a table. This allows a table to support mappings of string and integer keys to all data types.

The PS-algol interface to the persistent store is provided by the following three predeclared procedures.

a) create.database,
b) open.database and
c) commit.

1.3.5.1 create.database

This procedure allows a PS-algol program to create a new database. For example, to create a database called "Joe" with password "Bloggs" the following PS-algol could be used:

```
let rootTable = create.database( "Joe","Bloggs" )
```

The procedure creates the new database and then returns a table as the root object of the database. A table is represented by a PS-algol structure class so the result of the procedure is of type pntr.

1.3.5.2 open.database

This procedure allows a PS-algol program to access an existing database. The result of the procedure is the table that was returned by the create.database call that created the database. Therefore, to open the database "Joe" with password "Bloggs" the following PS-algol could be used:

```
let rootTable = open.database( "Joe","Bloggs","read" )
```

The third parameter to the open.database procedure indicates whether or not the database is to be updated. This parameter may be either "read" or "write" and is used to lock the database. A shared lock is requested for "read" and an exclusive lock is requested for "write". If the lock cannot be granted, the open fails and the open.database procedure returns an error message. This message is in the form of a structure of the structure class *error.record.* All three of the predeclared procedures use this mechanism to report errors.

1.3.5.3 commit

The commit procedure is used to invoke the commit algorithm that updates the persistent store; for example, a programmer may write:

```
let ok = commit()
```

The commit algorithm used by PS-algol ensures that any newly created objects that may be reachable from the root object of a database are made persistent. The commit algorithm only succeeds if all of the persistent objects that were changed came from databases that were opened using "write" as the lock parameter.

The following two short programs show how these procedures can be used to save and restore some data. The first program, shown in Figure 1.1, creates a database and then

saves an integer wrapped in a structure. The second program, shown in Figure 1.2, opens the database and retrieves the stored integer.

```
! a PS-algol structure class to hold an integer
structure intContainer( int aninteger )
! create a new database, db will be a table
let db = create.database( "Joe","Bloggs" )
! enter the integer into the database's table with the string key "int"
! s.enter is a PS-algol standard procedure used to operate on a table
s.enter( "int",db,intContainer( 3 ) )
! update the persistent store
let ok = commit()
```

Figure 1.1. A program to save an integer.

```
! a PS-algol structure class to hold an integer
structure intContainer( int aninteger )
! open the database for reading, db will be a table
let db = open.database( "Joe","Bloggs","read" )
! lookup the integer container in the database's table with the string key "int"
! s.lookup is a PS-algol standard procedure used to operate on a table
let p = s.lookup( "int",db )
! write out the integer value held in the structure
write p( aninteger )
```

Figure 1.2. A program to retrieve an integer.

The two programs can be easily modified to store any PS-algol data type simply by changing the type of the structure class field *aninteger*. Since the fields of a structure class may be of any type, these two programs demonstrate how any PS-algol data type may be made persistent.

1.4 System architectures

The PS-algol/ CPOMS system is one example of how the persistence abstraction can be realised. The main advantage of persistence is that it removes the discontinuities between the use of long and short term data. These discontinuities arose as a consequence of the evolution of computer architectures.

Originally, computer architectures were designed to compensate for the relative size and speed of main memory compared with long term storage devices. In particular, the relatively poor performance of early systems required as many operations as possible to be directly implemented in hardware to achieve efficiency. This made the hardware complex, expensive and the major component of every architecture.

Whenever new architectural mechanisms were developed, it was more cost effective to add new components to an existing architecture rather than design and develop a new architecture. As a result, a traditional computer system may provide virtual memory, a file system and a database system as three distinct components. The persistence abstraction can provide the facilities of these three architectural mechanisms within a single integrated system.

1.4.1 Software architectures

Modern computer systems are not subject to the same performance constraints as early systems. It is now possible to build a computer that can perform operations in software almost as fast as in hardware[pat81,pat82]. Consequently, computer architectures largely

composed of software can be efficiently implemented. Therefore, it is no longer necessary to develop a computer architecture specifically to suit particular hardware. This has two important consequences.

Firstly, a software architecture can be structured to suit the architectural mechanisms it requires, independently of the hardware on which it is implemented. This permits the many disparate components in a traditional architecture to be replaced by a smaller set of coherent mechanisms; for example, it is now possible to replace virtual memory, file systems and database systems by persistence.

The second advantage of adopting a software architecture is that it is possible to redesign the architecture without incurring the prohibitive costs, in time and money, that are associated with modifying hardware. This has the effect of allowing new architectural mechanisms to be integrated with the existing mechanisms, thereby avoiding the complexities of an ad-hoc extension to an existing architecture. Hence, a software architecture can provide a cost effective environment in which to conduct experiments with new computer architectures.

In order to exploit software architectures for persistence to the full, five areas have been identified and must be addressed[atk87b]:

a) complexity,
b) persistence,
c) system evolution,
d) protection and
e) concurrency.

1.4.1.1 Complexity

Complexity can be controlled by ensuring that an architecture conforms to a set of consistent rules. That is, as far as is possible, a rule should have no exceptions. Hence, if a rule applies to one component of a system it should apply to every component of a system.

An example of the effectiveness of this approach is provided by the Apple Macintosh system[app86a]. All applications developed for this system attempt to conform to a set of rules published by Apple. Therefore, if a certain set of key strokes and mouse clicks deletes text in one application then they have the same effect in another. Such rules allow new applications to be easily learned by users since the basic user interface is always the same. Hence, the consistent application of rules can be used to reduce the overall complexity of a system.

In order to avoid compromising the benefits of a set of consistent rules, new architectural mechanisms must be added with care. That is, a new rule should only be added to an existing system if it provides for encapsulation or abstraction over some existing rules. However, if a new rule introduces exceptions to the existing architecture rules, the total system architecture should be redesigned. This approach is feasible for two reasons. Firstly, the largely software nature of a new architecture is relatively easy to change and, secondly, the architecture to be changed should be relatively simple since it already conforms to a set of consistent rules.

1.4.1.2 Persistence

Persistence is a necessary feature of a software architecture since it separates the use of data from its storage. This allows the data structures best suited to modelling a problem's data to be chosen without regard to how the data is stored.

Persistence has the further advantage of reducing the overall complexity of a system by allowing all data to be manipulated by the same programming language constructs. Therefore, the components of a software architecture can be composed using the same mechanisms as those used to compose applications. A consequence of this is that the traditional division between system and application software is eliminated.

1.4.1.3 System evolution

An important feature of any architecture is its ability to adapt as new requirements are made of it. To support this, it is desirable to allow system components to be composed in new ways. This requires an architecture to composed of reusable components in such a way that individual components may be replaced by new improved versions.

Many of the possible approaches to this problem work well for small quantities of data but are not suitable for large scale systems. For example, all the components of a system may be recorded in a central dictionary that describes how each component may be used. Such a system would be unsuitable in a distributed environment since the composition of components would require a remote access to the dictionary. In a very large system, this could result in a significant performance overhead. Hence, a solution to this problem is needed that will work for large, possibly distributed, systems.

1.4.1.4 Protection

The protection of data in a system is necessary both to prevent the data being misused and to avoid it being lost as the result of a failure. Misuse can be prevented by the consistent application of protection mechanisms, such as type checking, to the entire architecture. Therefore, not only is a user's data subject to type checking, but so is the composition of applications. Furthermore, in a persistent system the type checking also applies to the composition of the architecture's components.

A stable storage mechanism must be provided to avoid data loss as a result of a failure. In order to avoid added complexity, this can be provided transparently by persistence. As described above, the minimum level of transaction mechanism provided by a persistent store is the updating of stable storage.

1.4.1.5 Concurrency

An architecture should support concurrency for at least three reasons. Firstly, the software architectures described so far, support the use of the appropriate data structures for modelling a problem's data. Therefore, it is also desirable to provide concurrency to support modelling a problem's behaviour. Another reason for providing concurrency is that in any large system data sharing is essential. For example, in a CAD system, two designers may be modifying their own designs, but wish to share a common pool of design data. The third reason is to support distribution. In a distributed system each node in a network operates concurrently. Therefore, if several nodes are combined to form a single distributed system, a mechanism must be provided to both describe and control their interactions.

1.4.2 A new software architecture

The structure of a software architecture does not conveniently divide into system and application software. In fact, the terms *system software* and *application software* are dependent on how a system is viewed. For example, the developer of a system may consider the programming tools he has developed to be his applications, whereas a user of a system may consider the programming tools as part of the system. To reflect this, the structure of a software architecture can be divided into the following four architectural domains[atk87a]:

a) the applications domain,

b) the language domain,

c) the system building domain and

d) the store domain.

The domains are not mutually exclusive nor are they the only way to classify the groups of activities and technologies that form a software architecture. In the same way as the terms system software and application software may apply to the same software, a particular activity or technology can belong to more than one domain.

1.4.2.1 The applications domain

The applications domain is composed of facilities specifically designed to support the production of applications. These facilities may include support for suitable programming methodologies, the provision of generic tool sets and libraries of reusable components. In addition to the facilities particular to this domain, an applications programmer also has access to all the other tools provided by the architecture and its implementation. The applications developed within this domain may also access these tools.

1.4.2.2 The language domain

The language domain provides the persistent programming languages that are used to implement applications in the applications domain. The language domain is intended to support all the programming activities in the system. Therefore, the languages must support the easy development of data models, generic tool sets, reusable components and persistent object stores. In particular, the languages map a system's data onto a stable store via suitable binding mechanisms, suitable type systems and suitable naming schemes.

1.4.2.3 The system building domain

The system building domain supports the construction of the persistent languages and the persistent environment. The major facilities of this domain may include compiler componentry, abstract program graphs, the merging of compilation and execution, support for abstract machine design and demand driven optimisation.

A problem with the facilities of the system building domain is that they are sometimes required to perform otherwise illegal operations. For example, a compiler component may be required to generate an executable procedure from a machine code representation. Clearly, this type of operation must be strictly controlled and hidden from the rest of the architecture.

1.4.2.4 The store domain

A major use of computers is as storage devices, a large proportion of system building is concerned with constructing and organising stores. Therefore, a large proportion of programmers will wish to use the architecture and programming languages to visualise a store and specialise it to their particular needs.

Within a persistent system, the persistent store is viewed as an infinite store of strongly typed objects. In particular, the implementation of the persistent store is not visible to a programmer. Therefore, the programmer can only operate on the persistent store through the type system and contextual naming mechanisms of a programming language.

The store domain provides a persistent store that can support any data types that may be defined in one of the architecture's programming languages. The support provided for certain data types may make use of special purpose hardware, data encryption techniques or data compression techniques. However, the provision of such support will never be visible to a user of the persistent store.

1.5 Type secure persistent object stores

The store domain must provide a type secure persistent object store upon which the other three architectural domains may be constructed. The technical problems involved in providing such a store are concerned with how to support a type secure store and how to support a persistent store.

1.5.1 Type secure stores

The provision of a type secure store requires that all operations on that store conform to a particular set of type rules. This may be performed within the store as a program is executed or within the compiler as a program is compiled.

The application of type rules within a store requires the store to have some knowledge of the type system it is supporting. That is, the store must be specifically designed to support the architecture's programming languages. This has the advantage of allowing the store to provide specific support for every data type. For example, raster images may be stored in a compressed form or on special purpose hardware. However, the store may not be able to support data types constructed within a different type system. This problem may arise when a new programming language is added to the architecture or if the store is used within a distributed system. In either case, the store would need to be redesigned to support the new data types.

The application of type rules by a compiler allows a store to be implemented independently of an architecture's programming languages. It is sufficient for the store to provide objects that identify references to other objects and for the organisation and interpretation of an object to be left to the compiler. This allows a store to support new programming languages without alteration. However, such a primitive store has no knowledge of the type system being used and is unable to provide special support for individual data types. For example,

it could not store large objects such as raster images in a specially compressed format or provide special purpose hardware to operate on them. This disadvantage may be partly compensated for by the compiler since, if it is generating machine specific code, it can take advantage of any special purpose hardware a machine may provide.

1.5.2 Persistent stores

The persistence abstraction can be extended in order to abstract over all the physical properties of data. Therefore, the implementation of a persistent store must be transparent to its users. That is, the size, speed and components of a persistent store are all hidden from a user by the persistence abstraction. As a consequence of this, a persistent store has certain perceived attributes such as unbounded size, infinite speed and stability. The main technical problems encountered when constructing a persistent store involve the simulation of these perceived attributes.

1.5.2.1 Store size

The ideal persistent store is of unbounded size. If it were not, its size could be determined by a program that created objects until the store was full. This would violate the persistence abstraction. Unbounded size can be simulated by using mechanisms such as garbage collectors and stacks to reclaim unused space. However, the particular choice of mechanisms is dependent on the type of computation being performed and the scale of the persistent store. For example, a persistent store supporting first class procedures may not be able to use a stack allocation scheme for recording all procedure activations. Similarly, an extremely large persistent store that is in continual use, may be constrained to use a garbage collector that operates concurrently with the store's users.

1.5.2.2 Store speed

The conceptual requirement on a store is that it is infinitely fast but, technologically, this is not possible. The only solution to this problem is to divide the persistent store into

components that can operate concurrently. In this way, the total performance of the store may be increased although it can never approach the ideal performance.

1.5.2.3 Store stability

The components of the persistent store are hidden so any failures must also be hidden. There are a wide variety of techniques that can be employed to simulate stable storage, but they all use the basic technique of maintaining more than one copy of the persistent store on non-volatile media. This may be performed by employing a strategy for regularly dumping the persistent store or always maintaining multiple copies of the persistent store on different disks. However, no stability mechanism can guarantee total stability, since all the copies of a persistent store may be subject to simultaneous damage.

1.6 Overview

The remainder of this thesis is concerned with the architectural support required to provide a type secure persistent object store. In particular, it describes the development of such a store as part of an architecture to support experiments in concurrency, transactions and distribution.

The architecture of the store has been designed to provide each distinct architectural mechanism as a separate architecture layer that must conform to a specified interface. The motivation for this design is two-fold. Firstly, the particular choice of layers greatly simplifies the implementation of the resulting architecture. For example, in a traditional computer system, a segmentation mechanism may be required to provide a large address space, organise the address space into logical objects and also provide a protection mechanism for the address space. In the layered architecture these three mechanisms are provided by three separate layers each of which need only implement a single mechanism.

The second motivation for the layered design is that it can support experimental architecture implementations. This is achieved by ensuring each layer conforms to its specified interface. Consequently, it is possible to experiment with the implementation of an individual layer without affecting the implementation of the remaining architecture layers. A further benefit of the layered design is that the reduced complexity also reduces the cost of conducting experiments. Thus, the layered architecture is a convenient vehicle for experimenting with the implementation of a persistent object store.

Chapter 2 presents a short review of the development of the various techniques that can be employed in constructing a type secure persistent store. These include techniques that can be employed to provide a type secure store, to simulate a store of unbounded size, to simulate an infinitely fast store and to simulate a stable store. It concludes by presenting three example systems that demonstrate how a selection of these techniques can be combined to construct a type secure persistent store.

This is followed in Chapter 3 by a description of the CPOMS. The CPOMS was designed and implemented as an experiment in constructing a persistent object store for use with the programming language PS-algol. As a result of analysing the advantages and disadvantages of the CPOMS, a layered architecture was designed. The evolution of this design is presented in Chapter 4.

A significant factor in the design of any computer architecture is the feasibility of implementing it. In Chapter 5, some of the practical issues relating to implementation are discussed. As a result, an optional layer is described that permits the complete layered architecture to be efficiently implemented with or without the support of special purpose hardware.

To demonstrate the feasibility of the architecture, Chapter 6 presents a description of the first implementation used by the programming language Napier[bro88a,mor88]. This is supplemented in Chapter 7 by a description of how the architecture could be used to support experiments with distribution. To conclude the description of the layered architecture, Chapter 8 presents a technique for tuning the architecture layers to reflect a particular choice of system configuration.

Finally, Chapter 9 presents a summary of the research undertaken and an indication of the directions for possible future research.

2 Implementation issues

The provision of a type secure persistent store must address the following issues:

a) type security,

b) store size,

c) store speed and

d) store stability.

Each of these issues present problems that have been well known for a number of years and to which many different solutions have been proposed. However, complete systems that attempt to provide a type secure persistent store are relatively recent and few in number[alb85a,lis84,that86]. The four issues will be discussed separately and their development illustrated by reference to a wide range of existing systems.

2.1 Type security

The provision of a type secure store requires that all programs operating on the store conform to some predefined rules. These rules are designed to control which data may be accessed by a program and how that data may be interpreted. The rules may be enforced in a number of ways, ranging from using special hardware to control access to certain areas of the store, to using a compiler to constrain the operations attempted by a program.

2.1.1 Separate address spaces

One of the basic techniques which may be adopted to control access to areas of store is the implementation of multiple address spaces. Initially, this technique was used to ensure that separate programs running on the same computer could not, accidentally or otherwise, corrupt each other. In addition to giving each program a separate address space, many computer architectures added two or more privilege levels[buc78,sch72,str69]. This provided a further level of protection to those parts of an operating system that have to be shared with a user's program. For example, the DEC VAX architecture[dec83] supports

separate address spaces for programs, together with the four privilege levels: user mode, supervisor mode, executive mode and kernel mode.

A further use of separate address spaces within traditional computer architectures is the division of a program's own address space. For example, a UNIX system will divide a program's address space between read only executable code and modifiable data[rit74]. This division is intended to prevent a program accidentally modifying the code it is executing.

2.1.2 Segmentation

The use of separate address spaces as described so far only provides a very coarse grain of memory protection. That is, it is unable to protect individual data objects within a program's data space. To overcome this deficiency and to permit the sharing of data between programs, the concept of *segmentation* was introduced[buc78,dal68,org73].

Segmentation allows individual data objects to be given their own address spaces together with a set of possible access rights such as *read*, *write* and *execute*. In most segmentation systems, the segments are identified by a fixed partitioning of the available address space. For example, a 32 bit address may be divided between a 16 bit segment number and a 16 bit offset within that segment.

To support the sharing of segments, individual programs identify those segments they wish to use with a symbolic name. When a program is executed, these names are used by a linkage mechanism to load an indirection table with the real address of each segment. In effect, each program has its own address space, dynamically composed from those segments it wishes to use. In the MULTICS system[dal68], this indirection table is known as the *segment descriptor table*. A separate data structure called the *known segment table* is used to support the mapping of a segment's symbolic name to a segment address.

In addition to these two data structures, every active procedure in a MULTICS system has an associated linkage segment. A linkage segment initially contains those symbolic names that a procedure may wish to use. When a symbolic name is first used, it is translated into an address in the process's address space. The resulting address is used to overwrite the symbolic name in the linkage segment, ensuring that the translation need only be performed once. In effect, the linkage segments define a separate address space for each active procedure. This address space is dynamically composed as the procedure executes, in the same way as a process's segment descriptor table.

The use of segmentation as in MULTICS has two major limitations. Firstly, the access rights available to each segment are only of a general nature and must apply to an entire segment. Clearly, this protection mechanism would be unsuitable for use with objects that were composed of both modifiable data and read only data. For example, if a segment were used to model a structured object consisting of a person's name and age, both the name and age would either be constant or changeable. It would not be possible for a person's name to be fixed for all time while their age was incremented annually.

The second limitation of this kind of segmentation mechanism is that there is no control over the construction of segment addresses. Thus, an erroneous procedure could construct the address of a segment it was not intended to access and then modify the segment. It should be noted that, although systems such as MULTICS do support multiple privilege levels, the address creation problem can still apply between segments at the same privilege level. Hence, some form of additional protection mechanism must be provided to control the manipulation of segment addresses.

2.1.3 Capabilities

Capabilities[cos74,fab73] were introduced to overcome the limitations of segment based protection mechanisms. The major difference between a capability system and a segmentation system is that in a capability system, the access rights for a segment are

recorded with each copy of the segment's address[coh76,eng74,nee74,rey79,ros85]. This allows different instances of a segment's address to have different access rights. Thus, a segment used to represent executable code may appear to be modifiable to a compiler or appear executable to a calling program.

To complement the distribution of access rights, a capability system includes a protection mechanism to control the allocation and communication of capabilities. That is, a capability system includes a mechanism to prevent an erroneous program addressing or modifying a segment it should not have access to. For example, in the IBM System 38[rey79], a pair of storage words are used to represent a capability. Each word forming the capability is tagged. To prevent a capability being illegally modified, the tag bits are automatically reset whenever a word is modified. Hence, a capability that has been illegally modified will no longer be recognised as a capability.

Another protection mechanism applied to capabilities is to only allow them to occupy explicitly delimited areas of memory. For example, in Hydra[wul74], each segment may be composed of two areas, a data area and a capability area. The forgery of capabilities is prevented by only allowing capabilities to be moved between capability areas or between a capability area and one of the machine's capability registers. It is not possible to copy a section of a data area into a capability area. These rules, enforced by Hydra, ensure that a capability can only be created or modified via the system's protection mechanism and not by an erroneous program.

In addition to controlling the communication of a capability, it may be possible to modify the access rights of a capability. This may occur if a program wishes to release a reference to a private data structure without allowing other programs to modify that data structure. This can be achieved by allowing access rights to be removed from a capability whenever it is copied. Thus, the capability for the private data structure would have its *update* right removed from the copies released by the program.

In general, it is undesirable to allow additional access rights to be added to a capability, since that would allow a program to invent access rights it was being denied. However, there is one case where it is necessary to add access rights, that is on a procedure call. In normal operation, a program will only have an *execute* right for the currently active procedure. For all other procedures, it will have a procedure call right, that is usually known as an *enter* capability. Thus, to be able to call a procedure and thereby execute it, the capability for that procedure must have the *execute* right added. This will be performed as one step when calling a procedure. A procedure call will also involve removing the *execute* right from the capability for the calling procedure.

In a sophisticated capability system, it may be possible to define the environments in which a capability may be used. To understand why this is necessary, consider a database that permits a browser to access some of its data structures. In a simple capability system, the browser would be able to export the capabilities for any data it encountered, thereby allowing the database's data to leak out. This would compromise any security the database was assumed to possess. To prevent such browser programs leaking information, some capability systems provide confinement or sealing/unsealing mechanisms, as outlined below.

The Hydra system supports confinement of capabilities[coh76] by providing access rights that allow a procedure to store a capability in an object other than its own activation record. Thus, the browser can be prevented from disclosing information by removing these additional rights from those capabilities the browser accesses. The capability system will then prevent the browser storing the capabilities in its own data structures or exporting them to other procedures.

The sealing/unsealing mechanism[wil82] works somewhat differently from the confinement mechanism. A sealed capability may be freely distributed throughout the system, but it

cannot be used until it is unsealed. The protection of the capability is achieved by controlling which procedures have access to the corresponding unsealing capability.

An alternative to the two previous protection mechanisms is to allow a capability to be revoked. That is, every copy of a capability, regardless of which access rights it may possess, is invalidated. This protection mechanism may be used to protect data by always handing out a capability that aliases the desired data. When the exported data has been used for its intended purpose, the alias capability can be revoked, thereby preventing any future misuse of the data.

It should be noted that in order to revoke a capability, it is necessary to be able to locate all copies of the capability. This ability can greatly complicate the design and implementation of a capability system. An example of a system that allows capabilities to be revoked is the Plessey System 250[eng74]. To avoid the additional complexity, systems such as Hydra support indirect addressing. A programmer may then make use of explicit indirections to simulate revoking a capability.

The type security mechanisms provided by segmentation and capability systems can be very sophisticated. However, they are all designed to be used with programs that may attempt to behave incorrectly. That is, the programs are able to directly manipulate their own data storage in an uncontrolled manner. These programs are not able to misinterpret segment addresses since these are explicitly protected by the capability system.

In some capability systems, the protection may be provided in part by software. To support this, the programs using the system may be required to conform to certain conventions. The enforcement of any conventions can never be assured if a program is written in assembly language. Thus, many capability systems do not allow assembly language to be used but rather enforce the use of one or more programming languages. For example, on the IBM System 38 programs are written in RPG-3[mye84]. Similarly, the MULTICS segmentation

system does not permit assembly language to be used, thereby preventing a program from accidentally altering segment addresses.

2.1.4 Compile time type security

Type security as provided by a capability system is mainly concerned with ensuring that a program operates on the correct data. Any control over the interpretation of data is usually limited to differentiating addresses, executable code and non-address data. This limited control may be extended by a high level programming language that provides a wide variety of different data types whose use can be checked at compile time.

Early programming languages, such as Fortran[ans78] and Cobol[sam62], provided facilities to help a programmer to structure data. As such, they were described as high level programming languages. However, initially their facilities were only of a limited nature and aimed at a specific class of applications. They also required a programmer to take great care in order to avoid unpredictable errors resulting from indexing errors or failures to correctly initialise data.

Later programming languages were designed to overcome some of these short comings by providing a greater variety of data types. For example, Pascal[wir73] provides a programmer with typed structures that can be used to model a wide range of complex data structures. Other examples of such programming languages include Algol-60[nau63], Algol-68[van69], BCPL[ric80] and C[ker78]. Although these languages can provide sophisticated type systems to control the use of data, they are still prone to spurious errors resulting from uninitialised data, indexing errors and so on.

Following the generation of programming languages that included Pascal, newer programming languages have been developed that use more sophisticated type systems and also reduce the potential for spurious errors. Some examples of these newer programming languages are Ada[ada83], CLU[lis81], Galileo[alb85b], KRC[tur82], ML[mil85],

S-algol[mor82], PS-algol[psa88] and SASL[tur79]. Each of these programming languages provide one or more of the following features:

a) abstract data types,
b) first class procedure values,
c) type inheritance,
d) the explicit initialisation of all data before it is used,
e) automatic bounds checking on indexing operations and
f) completely automatic storage management.

With respect to the type security of a programmer's data, the last three of these features are the most important. They are sufficient to guarantee that a program cannot accidentally or otherwise modify data outwith a programming language's type system. Within this thesis, these three features are assumed to characterise all high level programming languages. It is also assumed that a high level programming language does not allow explicit type coercions that alter the interpretation of a program's data. That is, an explicit type coercion is only allowed if the original data has a corresponding representation of the new type. For example, in PS-algol, an integer can be explicitly coerced into a floating point number since every integer has a corresponding floating point value, but not vice versa.

The use of a high level programming language that conforms to this definition can significantly affect the computer architecture required to support it. For example, if all programs operating within a computer architecture are written using a high level language, then they will each use the accessible data correctly. Thus, a protection mechanism based on separate address spaces may not be necessary. Indeed, the type systems of programming languages such as PS-algol do provide a subset of the protection facilities available within a capability system such as Hydra.

The benefits of eliminating separate address spaces from a computer architecture may be substantial. For example, the hardware required to implement an architecture need only support a single address space. As a result, a system would avoid the overheads involved in changing address space to execute a new process. Similarly, the hardware dedicated to controlling the manipulation of addresses could be eliminated. This would give rise to a two-fold benefit. Firstly, addressing operations could be performed without the need to validate access rights and, secondly, the overall complexity of the resulting architecture would be greatly reduced. Thus, an architecture relying on a high level language to provide the protection mechanisms has the potential to be both simple and efficient.

2.1.5 Object oriented programming

In recent years, the growing interest in object-oriented programming has led to the development of several high level language systems, CLU, Argus, Galileo and PS-algol. It should be noted that Smalltalk[gol83] has been excluded from this list since it allows explicit type coercion to alter the interpretation of a program's data. Therefore Smalltalk is not a high level language with respect to the description given above.

An interesting feature of the above four systems is the use of the system's programming language to implement the system's protection mechanisms. Thus, the implementations of each of these systems provide examples of how compile time type security can be used within an integrated system. This thesis describes the design of a layered software architecture that uses a high level language to implement the architecture's protection mechanisms.

2.2 Store size

Ideally we would like a store of unbounded size to match the unbounded nature of a programming language's ability to create objects. It is not possible to construct a store of unbounded size, but an extremely large store can be simulated. That is, by careful

management of the available storage, a store can be made to appear many times larger than it really is. There are two separate issues that must be addressed in simulating a store of unbounded size, the construction of a large store and the management of that large store.

2.2.1 Constructing a large store

The storage within a computer system is usually organised in some form of hierarchy consisting of processor registers, main memory and non-volatile disk storage. Each of these levels of storage has certain characteristics such as speed or size. The ideal store would operate like a computer's main memory as well as being large and non-volatile like the disk storage. Hence, the construction of a large store involves designing a mechanism that incorporates the best features of each storage level.

One technique that is used to simulate the ideal store is a form of *virtual memory*[den70]. That is, main memory is used as an efficient cache for a large area of disk storage that may be viewed as a large virtual main memory. The effectiveness of this simulation is dependent on the efficiency with which virtual addresses can be translated to main memory addresses and the quantity of data that must be moved between the disks and the main memory.

2.2.2 Virtual memory

Virtual memory was first introduced on the Atlas computer[fot61,kil62]. The Atlas had a one megaword address space that was partitioned into 2048 512-word units known as *pages*. There was a one to one mapping between pages and blocks on the disk storage. Thus, each virtual address could be mapped to a corresponding block on disk. The maintenance of the mapping was the responsibility of the host operating system.

The main memory of the Atlas was organised into page frames, of which there were 32. Each page frame had an associated virtual address register to record the virtual address of the page it contained. When a virtual address was used the address registers would be searched to find the page frame containing the desired virtual page. A main memory address

would then be formed by concatenating the page frame's address with the displacement within the virtual page, as is shown in Figure 2.1.

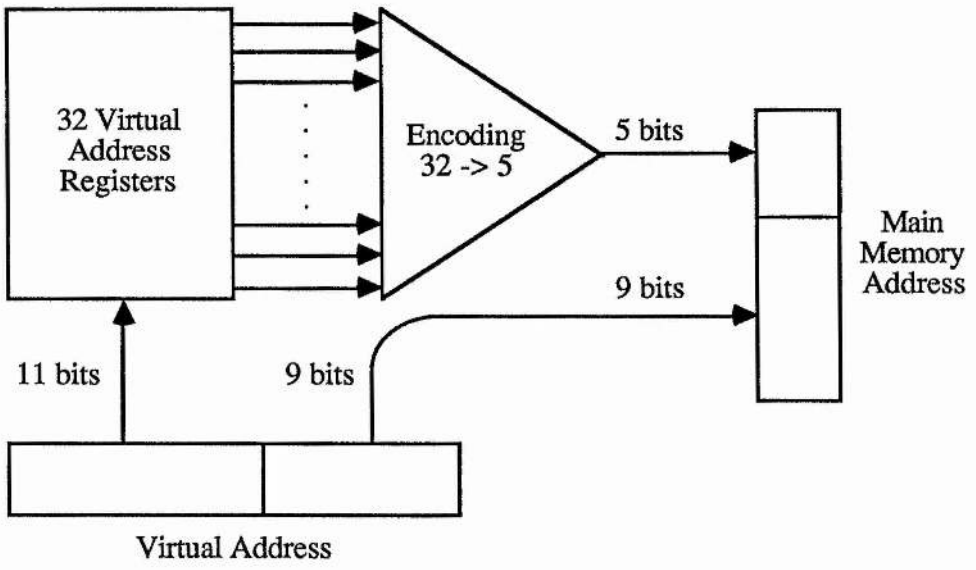


Figure 2.1. The Atlas virtual address translation hardware.

In the interests of efficiency, the virtual address registers were implemented in special purpose hardware that could search all the registers in parallel. Hence, a search was very efficient. Another feature of the search mechanism was that it generated an interrupt if the desired virtual page was not present. This interrupt was known as a *page fault*. When a page fault occurred, it was the responsibility of the host operating system to fetch the desired virtual page from the disk. This operation involved selecting a page frame to receive the new page, removing the page present in that page frame and updating the appropriate address register. Once the virtual page had been fetched, the addressing operation could be restarted.

The virtual addressing mechanism of the Atlas was very efficient because it used special purpose hardware. However, as the size of main memories increased, it became increasingly expensive to maintain an address register for every page of main memory. An alternative approach to address translation, adopted by other early systems, was to maintain an indexed page table with an entry for each virtual page. To support efficient address

translation, the page table was held in very fast memory. As the size of virtual address spaces increased, this scheme also became prohibitively expensive.

A further disadvantage of using special purpose hardware is the cost of setting up the special purpose hardware to perform address translations for a different virtual address space after context switching. For example, if the Atlas scheme was used on a main memory of a thousand pages then a change of virtual address space would require a thousand address registers to be updated. Clearly, this overhead would seriously degrade a system's overall performance if frequent context switches occurred.

2.2.3 Page tables

To reduce the dependence on special purpose hardware, later generations of virtual memory systems used indexed page tables held in virtual memory. To compensate for the relatively high cost of accessing these page tables, the most recently used mappings were cached in special purpose hardware. Consequently, the overall cost of address translation was reduced without incurring a large overhead in context switching.

This form of virtual memory scheme is suitable for use with operating systems that support a separate virtual address space for every running program, such as VAX/UNIX systems[bab81]. The paged page tables mechanism does have one significant disadvantage, namely the amount of main memory that must be occupied by page tables in order to support all the currently active virtual address spaces. A direct consequence of this is that the amount of main memory available to hold active data is reduced, resulting in an increased movement of data between main memory and disk. This problem is further aggravated if the virtual address spaces are very large.

A page table used to support a virtual address space contains mappings for each page of the page table. When a page table extends over several pages, it must be organised as several levels of page table. In a two-level page table, the first page contains address mappings for all the other pages of the page table, while the remaining pages contain the address mappings for the data pages. The first page of the page table must always be present in main memory to support the addressing of the page table. In addition, those pages containing the address mappings for active data pages must also be present. Thus, to address a data page via a two-level page table requires at least two pages of the page table to always be present in main memory. Furthermore, this overhead increases in proportion to the size of virtual address space in use. As a result, the number of active data pages that can be present in main memory is reduced, with a corresponding decrease in system performance.

2.2.4 Large virtual address spaces

The two virtual memory strategies described above present serious technical problems when applied to multiple large virtual address spaces. However, it should be noted that both strategies have certain desirable characteristics that may be adapted for use with segmented virtual address spaces. For example, the Atlas technique of maintaining mappings for only those pages present in main memory, is best suited to a single address space, whereas the use of page tables is suitable for use with several small address spaces.

In a segmentation system, each segment must be uniquely identified. This may be achieved by concatenating unique segment numbers, and addresses within segments, to form a single large virtual address space. The mapping of a segment onto the virtual address space may be performed when a segment is created or when it is first used. Once the mapping from segments to virtual addresses has been formed, a paging mechanism similar to that of the Atlas computer may be employed.

Examples of segmentation systems that utilise this technique are IBM system 38[ibm78b] and Monads[ros87]. These systems only record virtual address mappings for those pages

that are present in main memory. To reduce the total cost of implementing these systems the mappings are held in hash tables that are accessed by microprogram. The ideal solution, that of constructing special purpose hardware as used on the Atlas, would prove extremely expensive.

When a virtual address is translated, the address is hash encoded and the hash table searched to yield a physical page address. The structure of the hash table used by the Monads machine is shown in Figure 2.2. In the Monads machine, an address translation is performed as follows:

a) The virtual address is divided into a virtual page number and an offset within the page.
b) The virtual page number is then hash encoded to yield an entry in the hash table.
c) The virtual page number is compared with the page number held in the hash table entry.
d) If the two page numbers are the same, the physical address is formed by concatenating the physical page number held in the hash table entry with the offset found in step **a**.
e) If the two page numbers are different, the hash table entry's link field is inspected. The link field may identify an alternative hash table entry that must be searched.
f) If there is an alternative hash table entry, the address translation is continued from step **c**.
g) If no alternative entry is recorded the desired virtual page is not present in main memory. A page fault is then generated and handled by the operating system.

To aid efficiency, the hash table contains more entries than the number of pages that can be held in main memory. This is intended to reduce the number of collisions resulting from several virtual addresses having the same hash encoding. On the IBM system 38, the hash table has two entries for each page of main memory. It is estimated that for a uniform

distribution of virtual addresses, this will reduce the average number of hash table searches to 1.25 per address translation.

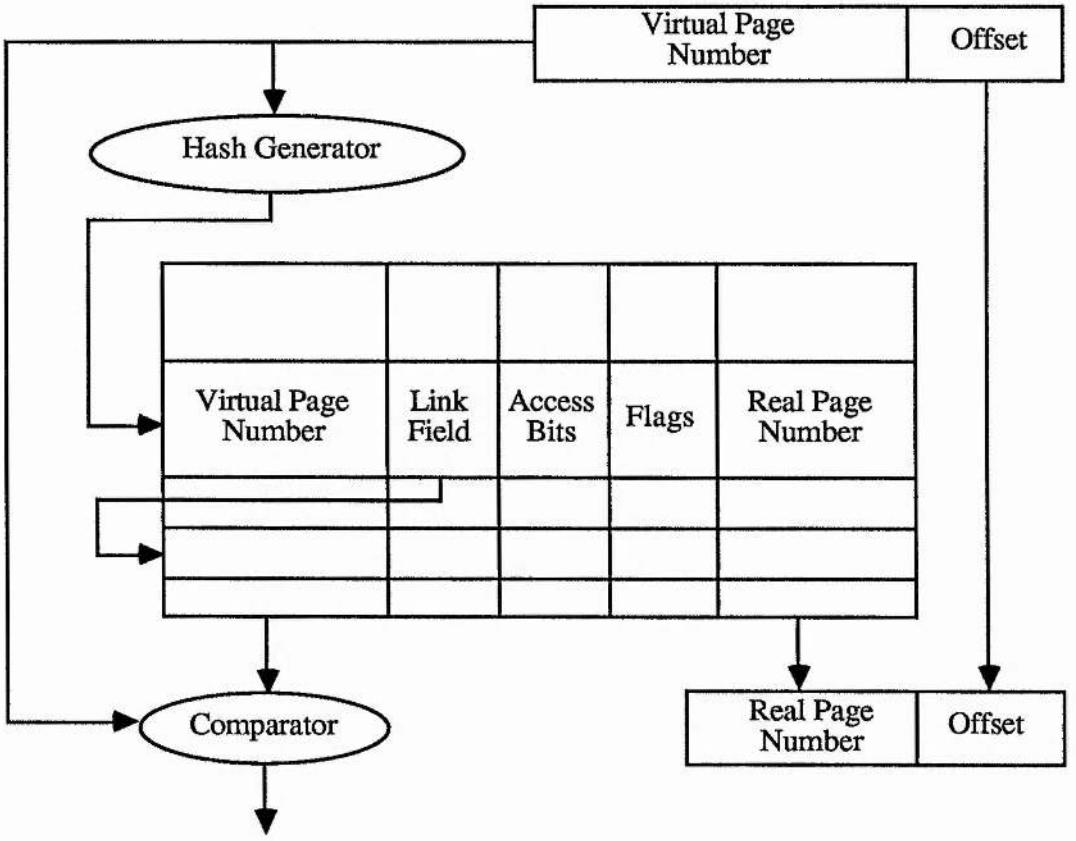


Figure 2.2. The Monads address translation unit.

An interesting feature of the virtual memory schemes used by Atlas, IBM system 38 and Monads is that the length of a virtual address has only a small effect on the total cost of the scheme. In the case of a hash table implementation, increasing the size of a virtual address only increases the size of an entry in the hash table. It would not significantly affect the searching of the hash table or the total time taken to translate a virtual address. Hence, the hash table based address translation technique is suitable for use with very large virtual address spaces.

The use of a virtual memory scheme similar to the Atlas requires the host operating system to be able to map a virtual address to a block of disk storage. To support this mapping, the Monads machine's operating system implements its segments using page tables. The advantage of using page tables is that the overhead involved is proportional to the size of a segment. Thus, a large overhead will only be incurred if a segment is large.

Another advantage of using page tables to support segmentation is the ability to reduce the fragmentation of disk storage. This can be important if the average size of a segment is significantly larger than the size of a page. For example, the Monads machine's segments do not need to be allocated a single contiguous area of disk storage. Instead they are allocated individual blocks of disk storage as they grow. In this way, a segment can grow as long as there are free storage blocks available. An alternative allocation strategy, based on contiguous storage areas, would only allow a segment to grow to the size of the largest contiguous area of free storage. However, such an allocation strategy would favour small segments.

When the average size of a segment is smaller than the size of a page, the majority of segments only occupy part of a page. A page may also contain some overhead for the segment's page table. Consequently, the proportion of active data to storage allocated may be quite small, resulting in large scale fragmentation of the available disk storage[ran69]. Hence, paged segments are best suited to a system whose average segment sizes are significantly larger than its page size.

The overheads involved in using page tables relate only to the actual size of a segment and not to the length of a virtual address. Consequently, the combination of the virtual memory techniques employed in segmentation systems such as Monads can be used to support extremely large virtual address spaces. A very large store may be constructed by simply providing sufficient disk storage onto which a very large virtual address space may be mapped.

2.2.5 Storage management

Two strategies may be employed to create the illusion of an unbounded store. Either a store may be constructed that can never be filled in the lifetime of a system, or storage may be recovered for reuse when it is no longer required. The virtual memory techniques that have been described above provide the technology to support a large quantity of data, but it is not yet feasible to attempt to construct a store that is too large to be filled. Hence the techniques developed to simulate a store of unbounded size are all based on recovering storage that is no longer required.

The techniques employed to reclaim storage are of one of two kinds. Either they rely on the explicit recovery of data storage or they use an automatic mechanism such as garbage collection.

The explicit recovery of data storage is employed by systems that can statically determine the lifetime of data. For example, if block retention is not supported, procedure activation records may be allocated on a stack. When a procedure returns to its caller, the stack can be retracted since the activation record is no longer required. Explicit recovery is also used by systems that support low level programming languages such as assembly language, C or Pascal. The implementation of these languages cannot support automatic garbage collection because it may not be possible to distinguish usable data from garbage. It would therefore be dangerous to attempt to perform an automatic garbage collection.

Automatic garbage collection of data requires data to be organised into self describing objects that identify references to other objects. To ensure that this condition is met, it is necessary to control the allocation of storage to data objects and to control the manipulation of object addresses. This can be achieved by either forcing all programs running in a system to be written in a high level programming language or by implementing a capability mechanism that prevents object addresses being misused. The benefit of using self describing data objects is that, given a set of root objects, a garbage collector can identify

other usable data objects by following object references. Any data objects not found by following object references can never be accessed and can therefore be deleted.

A wide range of garbage collection algorithms have been developed to suit the particular uses made of data storage[coh81]. For example, the garbage collection mechanism employed by the S-algol implementation uses a single list of free storage areas[mcc60]. This very simple storage organisation is sufficient to meet the needs of S-algol for two reasons. Firstly, data objects are not created at a very high rate. Consequently, the speed of allocation does not greatly affect the overall performance of the system. Similarly, the rate of object creation does not require frequent garbage collections to reclaim storage. Hence the cost of garbage collection has only a small effect on overall system performance.

Some uses of storage require objects to be created very efficiently. An example of this is given by PS-algol. In order to support block retention, the PS-algol implementation allocates a separate persistent object for each procedure activation instead of using a stack. In addition, individual blocks may also be represented by separate persistent objects if they form part of a procedure's static environment. As a result, an object may be created for every procedure call or block that is executed. For this reason, the allocation of data objects must be as efficient as possible since any delays will have a significant effect on the system's performance. To speed up storage allocation, PS-algol uses a compacting garbage collector developed by Morris[mor78]. This allows the available storage to be allocated from one contiguous area, thereby eliminating expensive searches of lists of free storage. However, it does involve a time consuming garbage collection when the available storage is exhausted. This cost is related to the amount of reachable data, but it can be offset by increasing the total amount of available storage.

More sophisticated garbage collection techniques have been developed that incorporate storage allocation mechanisms such as the *buddy system[*kno65]. The advantage of using these storage allocation techniques is that some data objects may be explicitly deleted

between garbage collections. The use of such techniques must be strictly controlled to ensure that reachable data objects are not deleted. In many cases, unreachable data may be determined by a static analysis of a program. For example, in a programming language that supports first class procedures, it may be possible to identify those procedures that do not require their activation records to be retained. When such a procedure returns to its caller, the procedure's activation record can be deleted.

The most significant factor affecting the performance of a garbage collector is the scale of the data on which it operates. This can manifest itself in a number of different ways. In the case of a compacting garbage collector, the entire storage area may need to be scanned more than once. Thus, garbage collecting an area of twice the size may result in more than four times the cost.

In order to avoid the potential costs of garbage collecting a very large store, a number of techniques have been developed. One technique developed for use with Lisp systems is known as *generation scavenging*[ung84]. The basis of the technique is that data objects are grouped according to their age. For example, the lowest addressed area of storage contains the oldest objects and the highest addressed area contains newly created objects. When the highest addressed area is full, it is garbage collected and compacted. To ensure that this garbage collection operates correctly, any references to new objects held in the other storage areas are taken into account. On completion of garbage collection, the highest addressed area may contain more usable data than a predefined threshold. When this occurs the garbage collection is repeated, but it will now also operate on the next oldest storage area. The repeated invocation of the garbage collector and the extension of the areas scanned is continued until all the storage areas conform to their respective thresholds.

The advantage of this technique is that a garbage collection of the highest addressed storage area will remove temporary objects without incurring the cost of garbage collecting the entire store. Furthermore, the compaction automatically moves the oldest objects towards the low

addressed areas of storage. Consequently, the division of the available storage into areas can be performed dynamically. Since the areas of storage are divided by the age of the objects they contain, the areas are known as *generations*.

Another common approach to large scale garbage collection is to perform incremental reclamation of storage. An example of this approach is the use of reference counting. This involves tagging each object with a count of the number of references to the object. Whenever an object address is overwritten, the object referenced by the old address will have its reference count decremented and the object referenced by the new address will have its reference count incremented. The storage allocated to an object is automatically reclaimed when the object's reference count is set to zero.

The advantage of a reference counting garbage collector is that the programs using the storage do not have to be suspended while unused storage is reclaimed. However, reference counting schemes require the overhead of a counter for every object that is created. A further disadvantage of reference counting is that it is not possible to automatically delete circular data structures. Hence, an object store implemented using reference counting may eventually be filled by unreachable, self referencing data structures.

2.2.6 Unique virtual addresses

There are a wide range of garbage collection techniques that may be successfully employed for a particular pattern of data use or scale of system. However, in certain situations, it may be necessary to introduce additional mechanisms to support the illusion of unbounded size. An example of this would be the implementation of a large scale distributed system. In such a system, it may be too costly to perform a garbage collection that scanned the entire distributed system. Hence, localised garbage collection techniques similar to the scavenging technique would have to be employed. This would allow the majority of unusable objects to be deleted automatically, with the exception of those that were reachable from a remote object. Eventually, individual systems within the distributed system may become full of

unreachable data. This spurious data would have to be deleted in order to allow the affected systems to function normally.

There are two techniques that can be employed when a system becomes full of unreachable data. One possibility is to extend the system's available storage by adding further disks. Depending on the organisation of the system, this may require the entire storage to be restructured, in addition to any costs involved in modifying the system's hardware. The total cost of this solution may prove prohibitive.

A second solution to the problem is to allow data objects to be explicitly deleted. This could be achieved by asking users to remove references to unnecessary data. A local garbage collection may then recover sufficient storage to allow the system to function correctly. Alternatively, the system could provide a system administrator with tools that can identify objects only referenced from a remote system. The system administrator would then have the responsibility of identifying and explicitly deleting any unreachable objects.

The deletion of an object requires that all references to an object to be invalidated. This can be achieved by either identifying all references to an object and then removing them or by performing all addressing via an indirection mechanism and then invalidating the object's indirection mapping.

Within a very large distributed system, it may be far too costly to scan the entire system to find all references to an object. Consequently, an indirect addressing mechanism must be used. In turn, this requires an object's address to be unique so that a deleted object's address will remain invalid. This can be achieved by making the number of available object addresses so large that they can never be exhausted. An example of a system that uses a combination of garbage collection and a very large number of addresses is given by Hydra[coh76].

2.3 Store speed

Although the provision of an infinitely fast persistent store is not technically feasible, it is possible to give the illusion of such a store by capitalising on potential parallelism.

One area of store design particularly suited to parallelism techniques is virtual address translation. For example, the majority of virtual address translation mechanisms make use of an associative store to support parallel searches for address mappings. On the VAX architecture this takes the form of a small address translation buffer that records the results of the last few translations.

Similar high speed address translation techniques have been used in the Atlas, Monads and IBM system 38. The Atlas uses an address translation register for every page frame of main memory to form an associative store that can be searched in parallel, as discussed in Section 2.2.2. The techniques used by the Monads and IBM system 38 are based on hash coding to search the associative store and are described in Section 2.2.4.

Another example of parallel address translation is provided by the SUN workstation[thak86]. The SUN uses dynamic RAM chips to implement its main memory. Access to this DRAM memory is performed in two steps. Firstly, the low order bits of an address are used to cycle a row of main memory and then the remaining address bits are used to cycle a column of main memory. To optimise access to main memory, the first step is performed in parallel with the translation of a virtual address by the SUN's memory management unit. A successful address translation will then result in the appropriate high order address bits being available for the second step of the main memory access. The success of this scheme depends on the number of rows in main memory corresponding to the page size used by the memory management unit.

Another area of store design that may permit parallelism is the particular choice of garbage collection technique. For example, an architecture may be designed in which the available storage was incrementally garbage collected in parallel with active programs. It may be advantageous to dedicate one or more processors to the task of performing this parallel garbage collection. Further parallelism could also be achieved by dedicating additional processors to other specific functions within the architecture. These functions may include analysing the current paging strategy, dynamically clustering data on disk storage and so on. An example of an architecture designed specifically with the intention of employing these parallelism techniques is given by Snyder[sny79].

2.4 Store stability

The persistence abstraction attempts to hide all the physical attributes of data. Consequently, the components of a persistent store are also hidden, requiring any failures in the components to be hidden. Therefore the persistent store is conceptually failure free that is, it is stable.

The potential failures that may occur within a store can be categorised as either being hard failures or soft failures. A hard failure is a failure that results in physical damage to the store, such as a head crash on a disk. A hard failure destroys data. In contrast, a soft failure is a failure that may cause a system to halt, possibly resulting in some minor corruption of data. In general, it will not result in the destruction of data. The provision of a stable store must address the issues of protecting data from the potential side effects of both hard and soft failures.

2.4.1 Hard failures

Hard failures may result in the destruction of data within a store. Consequently, the only way data can be protected from a hard failure is to maintain more than one copy of the store. The simplest solution to this problem is to make duplicate copies of the store at regular intervals. This technique is employed by the majority of operating systems and usually takes

the form of an archive on another storage device. For example, the operating systems used by traditional mainframe computers include sophisticated facilities designed specifically to coordinate the regular dumping of the system's disk storage[icl85].

The use of archive copies to preserve data may require a system to be shutdown for a considerable period of time while the copies are made. Similarly, restoring a system's data from an archive can also take a considerable amount of time. In general, these disadvantages are of little consequence because systems making sole use of this technique do not attempt to support extremely valuable data. Thus, a small loss of data as the result of a hard failure may be acceptable. Consequently, archive copies can be made infrequently, thereby reducing the overall effect on a system's performance while maintaining an acceptable level of protection.

The sole use of an archive to preserve data would be inappropriate for a large scale system that manages extremely valuable data. An example of such a system is a database system used by a bank to maintain its customer records. Clearly, any loss of data could result in substantial financial losses for the bank and would therefore be unacceptable. To overcome the special problems of very reliable systems, it is necessary to duplicate the store whenever an update occurs. This can be achieved by either maintaining more than one active copy of the system's data, each of which is updated in parallel or by maintaining a continuous log of all changes to the system.

The advantage of maintaining multiple active copies of a store is that a system can remain active while the store is being duplicated or restored. Hence, this technique is ideal for systems that are required to operate continuously. In contrast, the use of a log must be supported by a regular copying of the system's data by some mechanism such as an archive. When a hard failure occurs, the system's data is restored from the most recent archive copy. The changes recorded in the log are then applied to the store thereby reconstructing the state of the store as it was when the failure occurred.

The use of a log allows an archive strategy to be successfully used to protect a system containing extremely valuable data. However, the problem of protecting a log from hard failures is similar to that of protecting a continuously running system. That is, the log itself needs to be duplicated whenever it is updated. Hence, the log itself may be implemented by more than one active copy of its data.

The techniques employed to maintain several active copies of a store may be of differing levels of sophistication. For example, a possible technique that may be adopted is to poll the multiple copies of the store. That is, the current state of the store is determined by each copy of the store presenting its own version of the data. The version held by the majority of the copies would be assumed to be the correct version. This technique is widely used in distributed database systems[hol81,sch84,svo84]

Another technique that can be employed is to physically distribute the data between sites. Hence, a major catastrophe such as a fire or earthquake is unlikely to affect all the copies of the data. This technique is widely used by companies whose computer data is vital to their financial survival. A number of companies specialise in providing secure storage for valuable data, complemented by a recovery service that attempts to retrieve valuable data from damaged media such as disks.

In practice it is impossible to ensure that all copies of a system's data will remain free from corruption. That is, there is no such thing as a totally reliable store. However, given sufficient resources an arbitrarily high level of reliability can be achieved.

2.4.2 Soft failures

The protection mechanisms described above can be used to implement a stable store. That is, a store can be produced that, once modified, will retain the modification despite a hard failure. A potential disadvantage of a stable store is that it may not be in a self-consistent

state. That is, a failure may occur during a series of updates that prevents a logical operation from completing. As a result, the data held in the store will not be self-consistent.

To ensure that a store remains self-consistent, it is necessary to perform all updates to the store as some form of atomic transaction. That is, the modification either completes or it is totally undone. The basic mechanism for achieving a transaction is to maintain a record of which data has been changed, together with either its original value or its intended value. To ensure that the appropriate action can be taken on a failure, the record must be placed in stable storage before the update takes place.

The complexity of transaction mechanism provided by a system may be extremely varied. For example, consider a traditional database system such as IBM's System R[gra81]. System R supports several complex transactions operating concurrently, implemented by a combination of logging and checkpointing. Logging takes the form of recording all operations on stable store before the operation is performed. In addition to the normal operations, a record is kept of any checkpoints. Thus, when a failure occurs, System R can determine from the log how to restore its database to a self-consistent state.

To complement the logging mechanism, System R also provides a simple checkpointing mechanism that places the entire stable storage in a self-consistent state. The implementation of the checkpointing mechanism is based on shadow paging[lor73]. In normal operation, System R accesses its database via a paging mechanism. When a virtual page is modified, a copy of it is written to a new physical page and a mapping created between the two versions of the page. The effect of the checkpoint is to update the page mappings so that the modified version of each page is treated as the original version. The paging mechanism as described is continually forming a record of the changes to the system by preserving the original versions of each page.

In contrast to the complexities of a traditional database system, a single language system such as PS-algol or Galileo can adopt a much simpler transaction mechanism. For example, a PS-algol system supported by the PSPOMS implements its transactions via a shadow paging mechanism similar to that given by Challis[cha78].

Shadow paging is also used by the PS-algol/ Shrines system implemented under VAX/VMS[ros83]. Shrines operates by mapping a file holding the persistent store onto the virtual address space of a running program. This is achieved by directly manipulating the VMS page tables using a special purpose paging algorithm. The purpose of the paging algorithm is to ensure that when a page is to be modified, it is first copied and then the copy is modified. In this way, the original version of the persistent store is preserved while a new version is incrementally constructed. The transaction mechanism supported by Shrines allows the new version of the persistent store to become the original in a single atomic action.

An alternative approach, adopted by systems such as the PS-algol/ CPOMS system, is to record different versions of an object rather than different versions of a page. This has the advantage of permitting a finer granularity of transaction to be implemented than is possible using a simple paging mechanism. That is, two objects in the same page could have different versions and may be restored independently whereas in a paging system they would have to be restored together.

In these object based systems, the record of changed objects may be in one of two forms. Either it is a record of the original versions of the objects, known as a *before look*, or it is the new versions of the objects, known as an *after look*. A before look may be used to restore the store to a previous consistent state, whereas an after look may be used to complete the recorded updates and establish a new consistent state. In both cases, an update to the store is not performed until the entire before or after look is complete. Hence, the size

of a before or after look is dependent on the number of updates performed within each transaction.

The choice between a before or after look will depend on the particular use made of a store. For example, the CPOMS uses a before look since it was anticipated that updates to a database would contain a large proportion of new objects. Thus, a before look should be more efficient. Alternatively, the act of forming a before look may be expensive if additional accesses to a disk are necessary to retrieve the original value. Hence, the configuration of a system's buffering mechanisms may determine that an after look is more efficient. Ideally, a system using a before look or after look strategy should be able to switch between the two depending on the current use of the system.

2.5 Concurrency, distribution and transactions

The four properties of a type secure persistent store, type security, unbounded size, infinite speed and stability, are supported in part by Argus, Galileo and the Texas Instruments Persistent Memory System[that86]. The way in which each of these systems relates to the four properties is discussed below.

The Argus system provides a type secure store by forcing all programs to be written in Argus, which is itself a derivative of CLU. Argus is a high level programming language that supports a wide range of data types, including abstract data types and atomic objects. The store model supported by Argus relies on a garbage collector to recover unusable store. Hence, it also provides a mechanism for simulating a store of unbounded size. The stability mechanisms within Argus are based on explicit stable objects that are used to determine which objects are persistent. This is complemented by a sophisticated transaction mechanism that features sub-transaction and nested top-level transactions. Finally, Argus is designed to support distributed concurrent programs, thereby providing a mechanism for improving a system's performance and resilience.

Galileo is another single programming language system that utilises the language's type system as its main protection mechanism. Other features of the Galileo system are a shadow paged virtual memory system similar to that of IBM's System R. This stability mechanism is complemented by a logging mechanism to support user level transactions. Galileo is a high level programming language and its implementation includes a garbage collector. Hence, the simulation of unbounded store size is provided by the garbage collector. To date, Galileo is only available in the form of a single user system.

The Texas Instruments Persistent Memory System follows the example of Galileo, in that it is formed from a set of layers based on a stable virtual memory. It also utilises a logging mechanism to provide special resilient objects that may survive beyond the last checkpoint of the system. That is, a system failure may restore the virtual memory to its state at the last checkpoint, with the exception of the resilient objects. The system is designed for use with Lisp on the Texas Instruments Explorer. As such the protection of objects within the store is the responsibility of the Lisp system. In common with other Lisp systems the Persistent Memory System incorporates a garbage collector in order to simulate a very large store. The Persistent Memory System is currently being modified for use within a network of Explorer computers.

2.6 Conclusions

The three example systems in the last section each attempt to provide a type secure persistent store. However, the Argus and Texas Instruments systems both fall short of the ideal system since they allow some of the physical properties of data to be visible. In Argus, this includes whether the data is stable and where the data is physically located in a network. Similarly, the Texas Instruments system allows a programmer to distinguish objects that are more resilient than others. This aspect of the two systems violates the persistence abstraction and would be hidden in an ideal system. Despite these criticisms, the three example systems each provide a good approximation to the ideal type secure persistent store.

The remainder of this thesis presents the development of a layered architecture that will support experimental implementations of an ideal persistent system. The structure of the layered architecture has been designed for use with a high level programming language that provides the resulting system's protection mechanisms, as discussed in Section 2.1. Furthermore, the interfaces to the individual architectural layers have been organised in such a way that the layer implementations may be freely changed independently of each other. Thus, the illusion of unbounded space can be provided by any one of the techniques described in Section 2.2, without exerting undue influence on the provision of stable storage using the techniques described in Section 2.4, or any attempts to provide an illusion of infinite speed, as described in Section 2.3.

It is envisaged that any system constructed from the layered architecture should be able to meet the requirements of an ideal persistent system. That is, the resulting persistent system should be based on a type secure stable store whose physical properties are completely hidden from the programmer.

3 The CPOMS

The CPOMS is a persistent object management system written in C and is part of the PS-algol interpreter[psa85]. Its purpose is to provide the interpreter with an implementation of the kind of transactionally secure persistent store described in Chapter 1. It was designed as an experiment in implementing such a store to investigate the interaction between the various components of a persistent system. These components include a programming language, a type system, a transaction mechanism, a concurrency mechanism and a stable persistent store.

The persistent store implemented by the CPOMS was originally designed for use with applications that operate on several large disjoint pieces of data, but only need to be able to update one of them. To support this the persistent store is organised into units called *databases*.

There is a copy of the PS-algol interpreter for every active PS-algol program. Since the CPOMS is part of the PS-algol interpreter, there is also a copy of it for each active PS-algol program. Therefore the databases, which are passive, are operated on by multiple copies of the CPOMS, as shown in Figure 3.1. To ensure that PS-algol programs do not interfere with each other, a database may be accessed by only one program at a time if it is to be updated, or by several programs if it is only to be read. This is achieved by locking a database when it is first accessed, to indicate its intended use. To complement this, updates to the persistent store are performed only if all the databases that a program wishes to change are locked for updating.

In addition to controlling the sharing of data, the databases also provide the mechanism for identifying persistent data. That is, each database is used as a root of persistence, so that any object reachable from the first object in a database persists. The first object in a database is known as the *root object*.

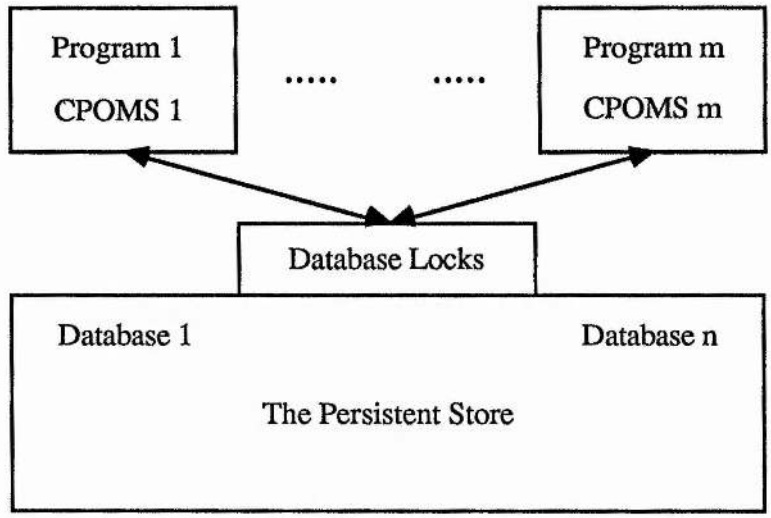


Figure 3.1. The interaction between the CPOMS and the persistent store.

To assist the organisation of the persistent store, the databases are identified by unique names. This naming facility is complemented by password protection to allow individual users to restrict access to private data. Thus, to access the persistent store, a program must identify a named database and supply the appropriate password. Once the named database has been identified and the password checked, the address of the database's root object is returned. This allows all data in the persistent store reachable from this root object to be accessed by the program.

To ensure that the data held in the persistent store remains self-consistent, a transaction mechanism is provided to control the updating of databases. This mechanism works by giving each program using the persistent store a copy of each object that it accesses. A commit algorithm may be invoked by a program to copy any objects it has changed back to their databases. This algorithm is structured so that either all the changed objects are written back or none of them are. In addition, checks are made to ensure that no objects are copied back to a database unless the program invoking the commit had opened the database with the intention of updating it.

The persistent store is accessed by a PS-algol program using a set of interface procedures provided by the CPOMS. These procedures, *open.database*, *create.database* and *commit*, allow a program to open a named database indicating whether or not it is to be updated, to create a new database or to invoke the commit algorithm, respectively. Since the address of the root object in a database is returned when the database is opened, no explicit interface need be provided to access individual objects. This can be performed automatically when an object is dereferenced. Hence, a PS-algol program can access individual objects by simply following object addresses starting from the database's root object.

3.1 Addressing persistent objects

The persistent store as supported by the CPOMS is logically divided into databases that are implemented as physically distinct units. This allows databases to be updated independently in a transactionally secure manner and yields a simple model of locking.

The commit algorithm must be able to identify which objects belong to a database. This may be achieved by either recording a reference to a database for each object, or by recording a range of object addresses for each database. The second method was chosen since it allows the mapping between an object and a database to be very simple.

A PS-algol program using the persistent store is unable to determine in which database an object is stored and may introduce inter database references between objects in different databases. To support arbitrary inter database references, all persistent objects have unique addresses within the persistent store.

To create unique persistent store addresses, the address spaces allocated to each database are divided into partitions of a larger address space. The allocation of the larger address space between databases presents the same problem as deciding how large to make segments in a segmentation system. The usual technique is to select a fixed division of an address between a segment number and an address within a segment. Such a partitioning of the address space

either allows a large number of small segments or a small number of large segments. This technique is unable to deal with a large number of small segments mixed with a few very large segments. To overcome this problem, the CPOMS divides its address space into relatively small fixed size partitions but allows a database to be allocated as many partitions as it requires. Therefore, the address space of the persistent store can accommodate a large number of small databases at the same time as a few very large databases.

The allocation of a partition to a database, is performed by mapping the partition onto a range of addresses that are local to the database. For convenience, a persistent store address is divided into two parts, a partition number and an address within the partition. The addresses within a partition are in a one to one mapping with the local addresses allocated to the partition.

Within a database, the CPOMS supports the use of indirect addressing. Each local address is an index into an indirection table that records the location of those objects owned by the database. Hence, local addresses, and the addresses within a partition, are known as object numbers. When a database is created it is allocated an area of private storage to contain the database's objects. When an object is added to the database the private storage is automatically extended.

The CPOMS uses indirect addressing in order to support garbage collection and to increase the potential size of the persistent store. The use of indirect addressing allows an object to be relocated within its database, without altering its unique address in the persistent store. This directly supports the efficient implementation of a compacting garbage collector. Without indirect addressing, the relocation of an object would alter its persistent store address, requiring all occurrences of the object's address to be identified and corrected. The cost of such an operation is directly proportional to the size of a store. Therefore, it would not be feasible to use a compacting garbage collector for a large persistent store without the use of indirect addressing.

The total size of a persistent store is limited by the mapping of its address space onto the available storage. For example, if addresses are directly mapped to bytes of storage, a 32 bit address could be used to address a persistent store with up to 2^{32} bytes of data. Alternatively, if addresses are mapped to individual objects, a 32 bit address could be used to address a persistent store of 2^{32} objects. Thus, the use of object numbers to address objects can greatly increase the potential size of a persistent store for a given size of address space.

To interpret an address in the persistent store, the following five steps must be performed:

a) the address is separated into two parts, a partition number and an object number within the partition,
b) the address's partition is mapped to a database,
c) the address's partition is mapped to a range of object numbers within the database,
d) the address's object number is added to the first object number allocated to the partition to yield an object number within the database and
e) the resulting object number is mapped to the address of an object in the database's private storage.

3.2 Addressing persistent objects from the interpreter

The addresses that the CPOMS uses to identify objects have no interpretation outwith the persistent store. Similarly, the addresses that the PS-algol interpreter uses to identify objects have no interpretation outwith the interpreter's heap. Clearly, this presents a problem when interfacing the CPOMS to the interpreter. One solution is to use special hardware so that the interpreter's heap and the persistent store are the same. That is, address translation hardware could be used to interpret persistent store addresses as they are used. However, the CPOMS was designed for use without special hardware and relies on the PS-algol interpreter to explicitly invoke address translation.

The explicit use of address translation requires the interpreter to test addresses before they are used, to see if they are heap addresses or persistent store addresses. To allow persistent store addresses to be differentiated from heap addresses, all persistent store addresses are negative, whereas all heap addresses are positive. This allows the interpreter to differentiate addresses by a simple sign test.

A persistent store address must be translated into a heap address before it can be used. This is performed by copying the object being addressed from the persistent store into the interpreter's heap. Thereafter, the object can be addressed via the heap address of its copy.

3.2.1 The PIDLAM

To support the translation of persistent store addresses, the CPOMS maintains a data structure called the PIDLAM, **p**ersistent **id**entifier to **l**ocal **a**ddress **m**ap. The PIDLAM contains a mapping from persistent store address to heap address for every copied object; it is depicted in Figure 3.2. When a persistent store address is translated, the PIDLAM is first searched to see if a mapping exists for the address. If a mapping is found, the corresponding heap address is extracted. This prevents multiple copies of an object being made, since all copied objects have a mapping in the PIDLAM. However, if the PIDLAM does not contain a mapping for the address, the object being addressed is copied into the interpreter's heap to yield a heap address. A mapping for the persistent store address is then entered into the PIDLAM for future use.

Since the translation of persistent store addresses is expensive, the CPOMS employs certain rules to minimise the number of translations. These rules are designed specifically to suit the PS-algol abstract machine. The rules also have the effect of minimising the number of checks required to identify persistent store addresses. Examples of how this may be achieved are given by the following three rules.

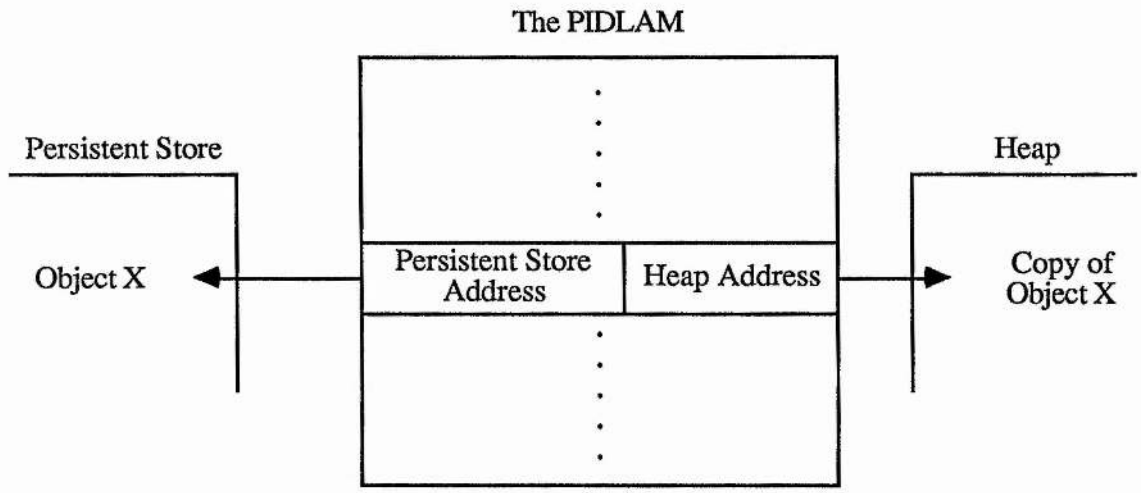


Figure 3.2. The role of the PIDLAM.

The first rule states that whenever a persistent store address is translated, the location holding it is overwritten with the corresponding heap address. This ensures that repeated use of a location holding a particular address only yields a persistent store address at the first use. To permit a persistent store address to be overwritten with a heap address, both kinds of addresses are the same size. This also allows objects containing addresses to have the same representation in the persistent store as they do in the PS-algol interpreter's heap.

The third rule states that all object addresses, placed on the stack of an executing procedure, must be heap addresses. Since the PS-algol abstract machine is stack oriented, this rule allows the many instructions that use the stack to assume the addresses they encounter are heap addresses. The effect of this rule is to reduce the number of checks for persistent store addresses, whereas the effect of the first rule was to reduce the number of address translations. All the other translation rules employed by the CPOMS have one of these two effects. A full description of the rules is given in the CPOMS manual[bro85].

3.2.2 The PTODI

The addressing of persistent objects requires several mappings; these mappings are supported by a data structure called the PTODI, the partition to database and index map. It

contains an entry for the partitions of all the databases that a PS-algol program is using. For each entry, there is a reference to the database that owns the partition and the first object number allocated to that partition. The use of the PTODI allows the CPOMS to efficiently perform the mappings necessary to address objects in the persistent store.

3.2.3 Summary of addressing

A summary of the addressing of a persistent object from within the interpreter can now be given. When a persistent store address is encountered, the PIDLAM is searched. If a mapping for the address is contained in the PIDLAM, it has a corresponding heap address. This heap address is used to overwrite the location that held the persistent store address. However, if there is no mapping in the PIDLAM, the object being addressed must be found in the persistent store, as shown in Figure 3.3. This is performed as follows:

a) The persistent store address is separated into two parts, a partition number and an object number within the partition.
b) The PTODI is indexed, using the partition number as the key, to find the database that owns the partition.
c) The PTODI is indexed, using the partition number as the key, to find the first object number allocated to the partition.
d) The object number within the partition is added to the first object number allocated to the partition, to yield the address's object number within the database.
e) The object number within the database is mapped to the address of the object in the database's private storage.
f) The object is copied from the database to free space in the interpreter's heap. This yields a heap address for the copied object.
g) A mapping from the persistent store address to the new heap address is entered into the PIDLAM. Any future use of the persistent store address will result in the heap address being found in the PIDLAM.

h) Once the PIDLAM has been updated, the complete set of address translation rules are recursively applied. The first of these causes the location holding the persistent store address to be overwritten with the new heap address.

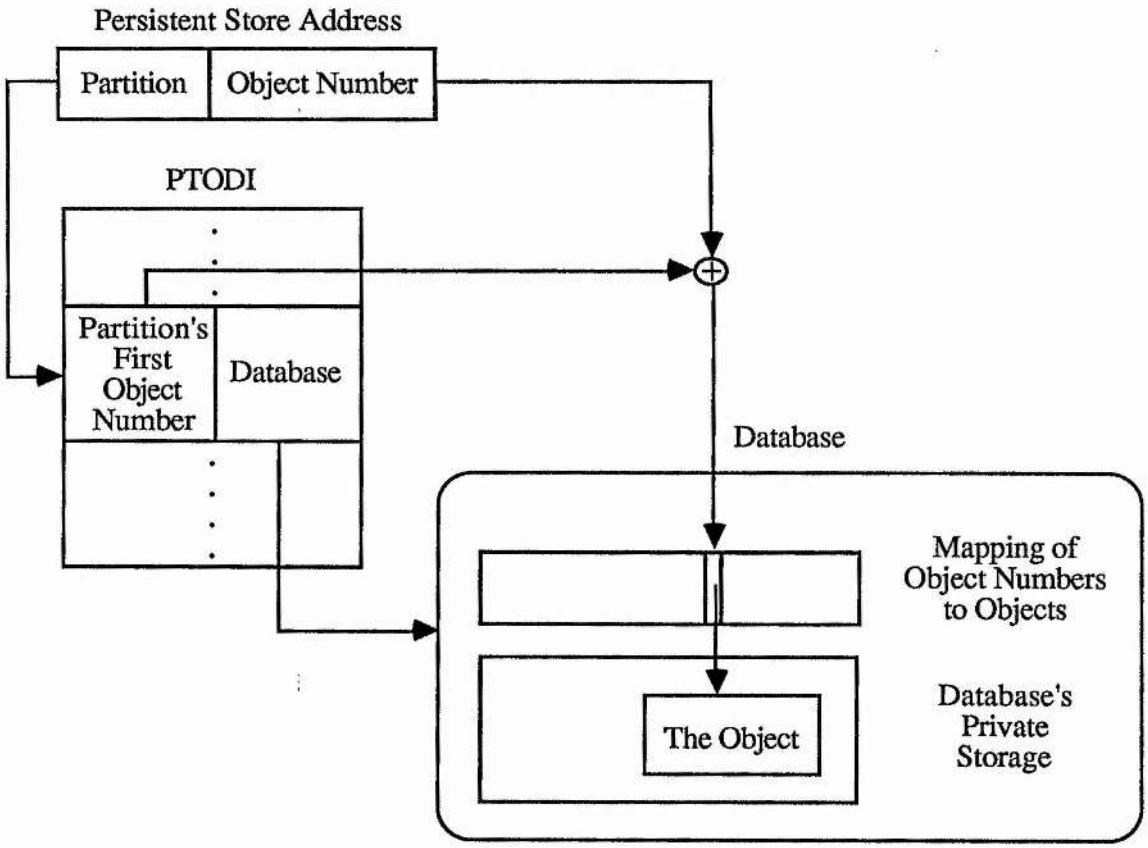


Figure 3.3. Addressing an object in the persistent store.

3.3 Committing changes to the persistent store

When a PS-algol program tries to access an object in the persistent store, the CPOMS copies the object into the program's heap. The CPOMS also provides a commit algorithm that allows modified objects to be copied back to the persistent store in a single atomic transaction. That is, all the objects are successfully copied or none of them are. The purpose of this section is to describe how the copying is performed.

3.3.1 Alternative commit algorithms

The first problem to be addressed by the implementation of commit is the choice of which data objects should be copied to the persistent store. Only persistent objects that have been changed need to be copied since persistent objects that have not been modified have the same value in a program's heap as they do in the persistent store. Objects that are reachable from a changed persistent object must also be copied.

It is not always desirable to commit every object described above. For example, a database opened for reading may contain data that should not be changed, objects from such a database could be ignored. Alternatively, changing objects from a database opened for reading might be considered an error, preventing any objects being committed. Another restriction that may be desired is to exclude objects that could never be reached from the root of a database. After the program calling the commit is finished, these objects could never be accessed.

Giving varying degrees of consideration to these problems, the following possible choices of objects were investigated:

a) Allow only one database to be opened for updating. Only commit objects reachable from the root of this database, ignoring any changed objects from other databases.
b) Commit every object reachable from the root of a database open for updating, but ignore any changed objects from databases open only for reading.
c) Commit every object reachable from the root of a database open for updating. If there are any changed objects from a database only open for reading, do not perform the commit.

d) Commit every object reachable from any changed object in the persistent store. If there are any changed objects from a database only open for reading, do not perform the commit.

It is not always possible for a PS-algol program to know in which database an object resides. Therefore, any commit algorithm that involves ignoring changed objects in read only databases will prevent a programmer predicting what a program will commit. If a program is attempting to enforce some high level consistency between disjoint pieces of data, this would invalidate the program's results. Hence, this highly undesirable feature led to options **a** and **b** being discarded.

Initially, every object copied from the persistent store is reachable from a root object. However, in the course of executing the PS-algol program, the imported path from an object to the root of a database may be broken. When this occurs the CPOMS must consult the persistent store to determine the object's reachability. Since there are no restrictions on inter database references, this may involve an exhaustive search of the persistent store if an object was unreachable. To avoid the potential cost of searching the entire persistent store, option **c** was also discarded.

As a result of the above problems, the CPOMS implements option **d**.

3.3.2 The CPOMS commit algorithm

The commit algorithm used by the CPOMS is composed of the following four steps:

a) identify all the copied objects that have been changed,
b) allocate persistent store addresses to new objects,
c) copy changed objects to the persistent store and
d) reset the change markers.

Every copied object has an entry in the PIDLAM. Also, every object created or changed by the PS-algol interpreter is marked as changed. Therefore, during a commit, the CPOMS can find all the changed objects copied from the persistent store by consulting the PIDLAM. However, the algorithm chosen for commit requires a check on whether any of the changed objects belong to a database that was only locked for reading.

To support this check, the CPOMS maintains a list of partitions that are allocated to the databases locked for updating. This list is consulted for every changed object with an entry in the PIDLAM. The commit algorithm terminates if it encounters an object whose persistent store address is not from a partition in this list.

The next step in the commit, is to allocate persistent store addresses to any new objects reachable from the changed objects. This is performed by a recursive address allocation routine starting from each changed object and terminating when an object with a persistent store address is encountered. Each application of the allocation routine is parameterised by two addresses, the heap address of the object to be inspected, the *child*, and the persistent store address of the object from which the heap address was obtained, the *parent*. The relationship between the parent and child objects is shown in Figure 3.4.

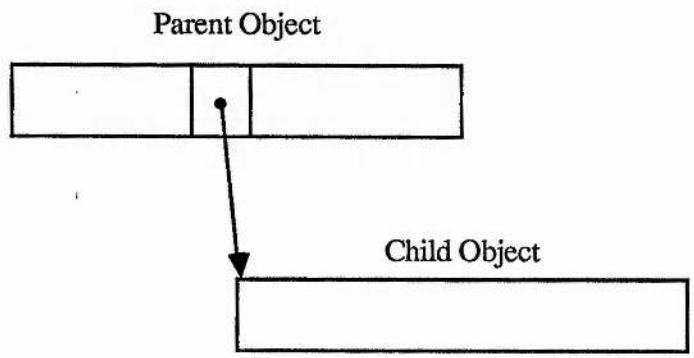


Figure 3.4. The relationship between the parent and child objects.

When the allocation routine encounters a child object that does not have a persistent store address, the object is allocated a persistent store address and an entry for the address is placed in the PIDLAM. In order to keep lists of objects together, the new address is allocated from the database containing the parent object. The address allocation is performed in the following five steps:

a) The object is allocated storage from the database's private storage.
b) If the database has no free object numbers, a new partition is allocated to the database. This extends the range of object numbers within the database.
c) The address of the allocated storage is recorded against a free object number in the database.
d) The object number is mapped onto a partition and an object number within the partition. The partition is the one that was allocated the range of object numbers containing the free object number.
e) Finally, the partition and the object number within the partition are combined to form a new persistent object address.

When a child object is encountered that has a persistent store address, the object's persistent store address is compared with that of the parent object. The reason for the comparison is that each database contains a list of other databases whose objects it may reference. The list is used when a database is opened and will be described later. If the comparison reveals that the two persistent store addresses come from different databases, an entry for the child object's database is placed in the list for the parent object's database. This reflects the fact that an object in the parent's database contains a reference to an object in the child's database.

When the allocation of persistent store addresses is complete, the PIDLAM contains an entry for every object with a persistent store address that must participate in the commit. The next step is to copy the changed objects to the persistent store.

During the copying of objects, all the heap addresses are converted to persistent store addresses. However, overwriting a heap address with a persistent store address may invalidate some of the assumptions permitted by the address translation rules. To avoid this problem, the heap addresses are converted in the I/O buffers as objects are being copied to their databases.

To optimise the addressing of the persistent objects being copied, the PIDLAM records the address of an object in a database's private storage. Therefore, that part of the addressing involved in looking up this address information can be performed without accessing the persistent store.

When all the changed objects in the PIDLAM have been copied to the persistent store, their changed markers are reset. They will not participate in a future commit unless they are subsequently modified. Any objects with changed markers that are not in the PIDLAM are not affected.

3.4 Opening a database

When a PS-algol program starts up, it has no means of directly addressing the persistent store; it only knows the names of the databases. Therefore, before any persistent objects can be accessed, the necessary address mappings must be constructed. In addition, a PS-algol program must also be given the addresses of the persistent objects. Since all objects in the persistent store are reachable from the root object of one of more databases, this can achieved by mapping a database name to the address of a root object. This operation is performed by opening a database. The act of opening a database also constructs those address mappings required to address any objects reachable from the database's root object.

3.4.1 The database directory

When a database is opened, the following three parameters must be specified:

a) the name of the database to be opened,

b) the database's password and

c) whether or not the database is to be updated.

The first step in opening a database is to map the database name to a physical database. To support this, the persistent store maintains a *database directory* that contains an entry for each database consisting of the database name, a password, the first partition allocated to the database, a record of any locks held on the database and a reference to the physical database.

The database to be opened is identified by searching the database directory for an entry with the desired name. Once an entry has been found, the database's password is checked to ensure that the database is being opened by a valid user.

The next step is to lock the database either for reading or updating. This is performed by comparing the requested lock with any locks already recorded against the database. As part of the comparison, a lock already held by the requesting program is ignored. An exclusive lock for updating a database can only be granted if the database is not locked, whereas a shared lock for reading a database can only be granted if the database is not locked for updating. While a database is being locked, the invocation of the CPOMS must have exclusive access to the database directory. This can be achieved by using the mutual exclusive primitives of the operating system on which the CPOMS is implemented.

The final use made of the database directory is to construct the address mappings for the first address partition allocated to the database. The mapping is made up of three parts, a database, an address partition and the first object number in the database allocated to that partition. All three of these parts can be deduced from the database directory. In particular,

the database is identified by its directory entry, the address partition is recorded in the directory entry and the first object number allocated to the partition is the first object number in the database. Once the address mapping has been completed, the CPOMS records it in the PTODI. All the persistent objects with an address in the database's first partition can now be addressed.

The database directory describes the address mapping for the first partition in a database. Any other partitions allocated to a database have their address mappings recorded within the database itself. To complete the opening of a database, it is necessary to construct these extra address mappings.

3.4.2 The database root object

The first object in every database is known as the *root object* since it is used as a root of persistence. That is, all objects reachable directly or indirectly from the root object will persist after a successful commit. In addition, the root object also records any housekeeping information required by the database, including a list of those partitions that are allocated to the database and a list of databases that may be referenced by objects within the database. To construct the extra address mappings, the list of partitions must be accessed.

By convention, the root object contains the first object number allocated to the partition containing the address of the list of partitions. The next step in opening a database is to construct the address mapping required to access the list of partitions and enter this mapping in the PTODI. Finally, the address mappings recorded by the list of partitions are entered in the PTODI. When this operation is complete, an address mapping will have been constructed for every object in the database.

3.4.3 Recursive opening of databases

When a database has been opened, the CPOMS has access to the address mappings for all the objects within the database. However, the database may contain the addresses of objects

in unopened databases. That is, the CPOMS may encounter persistent store addresses for which it has no valid address mappings. There are two alternative approaches to this problem: raise an error or open every reachable database.

The simplest solution is to indicate that an error has occurred when such an address is dereferenced. However, when the CPOMS was designed, this was unsatisfactory for two reasons. Firstly, PS-algol contains no means of identifying where an object is stored. That is, the databases only provide knowledge of the access path to an object. Consequently, it is not possible for a PS-algol program to determine in which database an object resides and therefore it cannot anticipate the error.

The second problem with raising an error is that PS-algol contained no exception handling mechanism that could deal with it. As a result, such an error would have caused a program to crash. The combination of these two problems meant that it would not always have been possible to anticipate if a program might crash, particularly if the programmer was not responsible for the databases the program used.

To avoid these problems, the CPOMS opens every reachable database. This operation is performed once all the address mappings for a database have been constructed, by recursively opening each database referenced by the list of reachable databases.

When a database is opened by a recursive open, the database name and password are not used. Instead, a direct reference is used to a database's entry in the database directory. The choice of lock for a recursive open is usually a lock for reading. However, when an unnamed database is opened, a lock for updating is requested. The reason for this is that every other database can be explicitly opened for updating by a user, whereas an unnamed database cannot.

3.4.4 Accessing objects in a database

The act of opening a database has so far only described how the address mappings are constructed. In addition to this, an open must provide a mechanism to allow a program to access the objects in a database. This is achieved by returning the address of a database's root object to the calling PS-algol program. The root object contains an address field that can be used by a PS-algol program to address a user's data. For convenience the root object contains the address of a PS-algol *table* with which a user can organise his data. In fact, the *open.database* procedure used by a PS-algol programmer is a PS-algol procedure which calls the CPOMS *open.database* and then returns the address of the table. This allows the table to be treated as the database's root object.

3.4.5 Summary of opening a database

When a database is opened, the following three parameters must be specified:

a) the name of the database to be opened,
b) the database's password and
c) whether or not the database is to be updated.

A summary of the steps involved in opening a database can now be given. The steps involved are as follows:

a) Search the database directory.

   The first step is to search the database directory for an entry with the specified database name. The open fails if there is no database with the specified name.

b) Check the password.

   The database's password is then compared with the specified password. The open fails if the passwords are not the same.

c) Lock the database.

A lock is then requested on the database. This is an exclusive lock if the database is to be updated, otherwise it is a shared lock. The open fails if the requested lock cannot be granted.

d) Construct the first partition's address mapping.

The address mapping for the database's first partition is then constructed using the information held in the database directory. The mapping is recorded in the PTODI so that the database's root object can be addressed.

e) Construct an address mapping to access the list of partitions.

The database contains an object that records the partitions allocated to the database. The address mapping for the partition containing this object is calculated using information held in the root object and entered into the PTODI.

f) Construct the remaining address mappings for the database.

An address mapping is then constructed for each partition allocated to the database and entered into the PTODI. Every object in the database can now be addressed.

g) Open all potentially reachable databases.

To ensure no addresses are encountered for which no address mappings are known, every database that may contain a reachable object is recursively opened. The initial open fails if any of the recursive opens fail.

h) Return the address of the user's data.

The final step is to return the address of the database's root object to the calling PS-algol program. A user's data can then be accessed by dereferencing the root object.

3.5 Creating a database

When a PS-algol/ CPOMS system is first initialised, it does not contain any databases and therefore does not provide any roots of persistence. The CPOMS includes a mechanism to create a new database and thereby introduce a new root of persistence. This operation

involves creating the database, allocating an address partition to the database and constructing an entry in the database directory.

3.5.1 The database creation

When a database is created, the following two parameters must be specified:

a) the name of the database to be created and
b) the database's password.

The first step in creating a database is to search for a free entry in the database directory. As part of the search, a check is made to ensure that the new database name is unique. If it is not, then the create fails. Once a free entry has been found, the database's name and password are entered into it.

Next, a free address partition is allocated to the new database and the new partition is recorded in the database's directory entry. Since this is the first partition allocated to the database, its first object is the database's root object. Hence, recording the partition in the database directory allows the root object's address to be calculated when the database is opened.

The next step is to create the physical database. This is performed by allocating an area of private storage for the new database and then creating two objects within the storage. The first object is the new database's root object and the second object contains a record of the new partition. To support *open.database*, the root object contains the address of the second object. The second object's address corresponds to the second object number in the new partition.

The final step in creating the database is to enter a reference to the physical database in the database directory. The new database and its entry in the database directory are then available for use by *open.database*.

3.5.2 Opening the new database

Once a new database has been successfully created, it is opened for updating to allow a user to insert data into the new database. The open is performed using the CPOMS *open.database*. *Create.database* then returns the address of the root object in the database to the calling PS-algol program.

The *create.database* procedure called by a PS-algol programmer is a PS-algol procedure that calls the CPOMS *create.database*. This indirection is present so that the address of a PS-algol table can be automatically inserted into the new database's root object. Hence, a PS-algol programmer can use the PS-algol *open.database* and *create.database* procedures and assume that every database has a table as its root object.

3.6 Garbage collection

The CPOMS simulates a store of unbounded size using two different garbage collection mechanisms. This is a consequence of the two different address spaces in which the CPOMS must operate: the persistent store and the heap of a PS-algol program.

3.6.1 A two level store

The heap of a PS-algol program and the persistent store have very different characteristics. For example, the persistent store will contain a very large number of potentially long lived objects, whereas a significant proportion of objects within a program's heap will be short lived. Furthermore, the potential size of the persistent store makes garbage collection an expensive though infrequent operation. In contrast, the small size of a program's heap permits garbage collection to be a relatively cheap but frequent operation.

The use of two address spaces has several advantages over a single address space for the purposes of garbage collection. Firstly, since a program's heap will be frequently garbage collected, the store allocated to short lived objects may quickly be reclaimed. Another advantage is that the garbage collection of a program's heap can be performed without interfering with any other programs. Furthermore, this is a relatively inexpensive operation since the garbage collection is performed in main memory and only operates on those objects that the program is using.

The use of the two address spaces also reduces the use made of the persistent store. That is, the persistent store is only updated by the commit algorithm. A feature of commit is that it only copies reachable objects to the persistent store. In particular, new objects are only copied if they are reachable from an object already present in the persistent store. Hence, the program invoking commit must have created these new objects with the intention of making them persist. Therefore, only a relatively small proportion of the objects created by a program will reach the persistent store, thereby reducing the speed with which the persistent store is filled. In turn, this reduces the frequency with which the persistent store must be garbage collected. This effect can be further enhanced if the invocation of commit is infrequent, so that fewer temporary objects are made persistent.

The disadvantage of the two store model is that two garbage collectors are required, one to garbage collect a program's heap and one to garbage collect the persistent store.

3.6.2 Garbage collecting a program's heap

The PS-algol interpreter provides its own garbage collector to operate over a program's heap. The garbage collector is designed to discard all objects that are not reachable from one of the interpreter's abstract registers. Therefore, it must be able to deal with persistent store addresses. This is achieved by two simple mechanisms.

Firstly, the garbage collector ignores all persistent store addresses it encounters. Hence, the garbage collector only identifies reachable objects that have heap addresses. This modification limits the garbage collector to operating within a program's heap.

The other modification is to use the PIDLAM as an additional starting point for identifying reachable objects. This ensures that no changed objects, copied from the persistent store, are discarded until such time as the commit algorithm is invoked. For example, if a persistent object is changed to reference a newly created object, the next commit must copy these two objects to the persistent store. The persistent object must be copied because it has been changed and the new object must be copied since it is reachable from a persistent object. However, it is possible that the two objects will not be reachable from the program when a commit is performed. Therefore, they must be retained even if they are not reachable. This is achieved by using the PIDLAM to identify reachable objects, since every persistent object copied into the heap is reachable from the PIDLAM.

3.6.3 Garbage collecting the persistent store

The CPOMS does not garbage collect the persistent store. Instead, a separate disk garbage collection program is provided with each PS-algol/ CPOMS system. This solution was chosen for the following reasons:

a) The CPOMS is part of the PS-algol interpreter.

   Since the CPOMS is part of the PS-algol interpreter, it must always share the persistent store with at least one active program. Therefore, there will be at least two views of the persistent store that must be reconciled at the end of a garbage collection. Separate views arise because a program using the persistent store has its own copies of those objects it has accessed.

b) The CPOMS cannot represent all the persistent store's address mappings.

The address mappings maintained by the CPOMS are designed to efficiently address a relatively small part of the persistent store's address space. As such, they contain certain extra information to allow the mappings to be searched efficiently. Consequently, the storage requirements necessary to address the entire persistent store are considerable. Also, since the address mappings are held in a program's heap, it may not always possible to allocate sufficient storage to construct all the address mappings.

c) The CPOMS cannot support changing the entire persistent store in one transaction.

An important feature of the persistent store is that it can always be restored to a self-consistent state. The CPOMS achieves this in three steps. Firstly, it copies those objects it wishes to change and changes the copies. The original objects are then duplicated on the disk so that they can be used to restore the persistent store. Finally, the original objects are overwritten by the changed copies. Since the objects are copied into a program's heap before being changed, the CPOMS may need to copy the entire persistent store into a program's heap during a garbage collection. However, the persistent store is very much larger than a program's heap, making this solution infeasible.

There are several advantages to providing the persistent store's garbage collector as a separate program. For example, the garbage collection algorithm used is not constrained to operate concurrently with the CPOMS or any running PS-algol programs. Also, the persistent store may be supported by several different garbage collectors, each of which is designed for a different size of store and pattern of use. This flexibility would not be possible if the garbage collection algorithm was an integral part of the CPOMS.

3.7 Implementation details

A detailed description of the CPOMS and its UNIX implementation can be found in the CPOMS manual[bro85]. This includes a description of how the persistent store is mapped onto the UNIX file system, a description of the commit algorithm's recovery mechanism and a complete description of how persistent store addresses can be efficiently translated by software. It also describes the error recovery mechanisms that are applied to the internal data structures such as the PIDLAM. An important feature of the CPOMS is that, as far as possible, it will attempt to recover from an error. This allows a PS-algol program to take corrective action or issue a suitable error message to a user. The class of errors that can be recovered from include failing to open a database because its locked and attempting to perform a commit when no databases have been opened for updating.

The garbage collector used by the PS-algol interpreter is described in Chapters 8 and 9 of the PS-algol abstract machine manual[psa85]. A garbage collection program for the PS-algol/ CPOMS system is given by the author[bro88b] and an alternative program is given by Campin[cam86].

4 A layered architecture

The CPOMS described in the previous chapter provides a transactionally secure persistent object store. This chapter describes a layered architecture that was designed to capitalise on the advantages of the CPOMS, while minimising its limitations.

4.1 The CPOMS

The CPOMS architecture has several inherent limitations. In particular, the databases around which the CPOMS is organised give rise to three particular problems.

The intention of introducing databases was to provide a mechanism to allow a user to structure a collection of data so that it could be effectively shared. However, the PS-algol model of a store provides no notion of how or where an object is stored. A database is thereby restricted to only providing an identifiable access path for an object. Thus, databases cannot be used to identify shared data as the user does not know in which database the data resides.

Another serious limitation of the CPOMS is that concurrency control is performed at the level of database locking. When a program locks a database for updating, it has exclusive access to the database. This prevents two programs concurrently updating disjoint pieces of data in the same database, even though the programmer may be aware of the potential concurrency. This particular problem is further aggravated by the recursive opening of databases. It is even possible that data known to be in separate databases may not be updated concurrently, due to an inter database reference that forces both databases to be opened together. The exclusive access required to update a database would prevent the concurrent updating of the two databases by separate programs.

The third problem with the CPOMS is that it only supports programs that manipulate a small part of the persistent store. Programs that use a persistent store can be divided into three behavioural groupings:

a) those programs that read and possibly update a small part of the persistent store,

b) those programs that may read the entire persistent store, but only update a small part, and

c) those programs that may read and update the entire persistent store.

The CPOMS is ideal for programs in group **a**, but is unable to deal with programs in groups **b** and **c**. The reason for this shortcoming is that the object addressing mechanism only collects objects from databases and never discards them. As a result, the heap of a PS-algol program is incrementally filled by a copy of the persistent store. Since the size of a program's heap is relatively small compared to the persistent store, the CPOMS may attempt to collect more objects than the heap can hold. When this happens, the PS-algol/ CPOMS system fails.

Programs in group **b** can be accommodated by discarding objects that have not been modified. This can be performed provided that the assumptions made by the address translation rules are preserved. Note this requires heap addresses to be overwritten with their corresponding persistent store addresses, as objects are discarded, to avoid dangling references. A CPOMS supporting this facility has been constructed[spa88].

Programs in group **c** are a lot more difficult to accommodate since they require changed objects to be discarded. This may be achieved either by writing changed objects back to their databases or by writing changed objects to a temporary store.

The first solution requires an incremental version of the commit algorithm to be implemented. Such an incremental commit would only be able to operate if all the databases containing the changed objects were open for updating. An object from a read only database could not be discarded, since this would cause the read only database to be modified. A further disadvantage of this solution is that the automatic recovery mechanisms would need

to be made much more sophisticated to cope with multiple pending commits. The CPOMS as described in the previous chapter is already very complex and the additional complexity required would be extremely costly to implement.

The alternative solution of using a temporary store is also expensive to implement. For example, the addressing and transaction mechanisms used by the CPOMS would need to be modified to allow for the correct version of an object being held in the temporary store. Another problem that may arise is that of allocating unique addresses to newly created objects that may be written to the temporary store. In effect, the use of a temporary store would require the CPOMS to incorporate an additional level of object management, with a corresponding increase in complexity.

4.1.1 Vertical structure

Apart from the limitations of the databases, the CPOMS is a very inflexible architecture for the purposes of experimentation. This is because the CPOMS architecture, and indeed the entire PS-algol/ CPOMS system, is vertically structured. Examples of this can be found in the format of the PS-algol heap objects, the CPOMS automatic recovery mechanisms and the allocation of reserved addresses.

The format of the PS-algol heap objects are specifically designed to suit the PS-algol abstract machine. They include strings, structures, stack frames, code vectors and raster images. Each of these object formats must be known to the PS-algol compiler to allow the correct abstract machine code to be generated. The abstract machine needs to know them so that it can manipulate the objects correctly. The garbage collectors operating on the abstract machine's heap and the persistent store need to know the formats so that they can identify the address fields within an object. Finally, the CPOMS needs to know the object formats so that it can apply the address translation rules and perform any specialised actions necessary to store certain objects.

Since the object formats must be known to every component of a PS-algol/ CPOMS system, it is extremely difficult to experiment with the formats. Not only must all the different components in a system be modified, but great care must be taken to identify all the potential side effects For example, does a new object format need to be saved by the CPOMS in a special way?

The vertical structure of the CPOMS is reflected in the way that every data structure the CPOMS uses must include a recovery mechanism. This is because every database operation causes some or all of these data structures to be modified by side effect. For example, opening a database may modify the PIDLAM, the PTODI and possibly the list of writeable partitions, even if it fails.

Furthermore, vertical structuring is demonstrated by the commit algorithm. As a commit progresses, it allocates new persistent store addresses that must be recorded in the PIDLAM. In turn, this may result in new address partitions being allocated to a database; these must be recorded in the PTODI and the list of writeable partitions. Consequently, the commit algorithm must coordinate the recovery mechanisms applied to the PIDLAM and PTODI with those applied to the databases. In addition, the commit algorithm must also coordinate all the disjoint components of the persistent store that are being modified. As a result of this large information flow, it would be impractical to separate the recovery mechanisms from a commit.

Finally, the allocation of reserved persistent store addresses by the CPOMS involves communication between the CPOMS, the PS-algol abstract machine and the PS-algol compiler. The compiler is involved since it provides the list of structure class descriptions that receive reserved addresses. It must ensure that the list is in a known order. The abstract machine is involved since it provides the primitive standard procedures for PS-algol, each of which receives a reserved address. It must ensure that only the correct standard procedures are provided. This particular example of the vertical structure of the PS-algol/ CPOMS

system is easy to change if a totally new system is being constructed. However, it causes considerable problems if an existing system is to be enhanced by the addition of new standard procedures or predeclared structure classes. At present, the only solution to this problem is to allocate sufficient dummy standard procedures for future expansion and not to give any extra structure trademarks reserved addresses.

4.1.2 Protection and addressing

Despite its many limitations, the PS-algol/ CPOMS system has proved successful in two areas, the protection of the persistent object store from incorrect programs and the small performance overhead of the object copying mechanism when combined with the address translation rules.

The protection of the persistent object store is not performed by the CPOMS. Instead, the PS-algol compiler is relied on to ensure that all programs behave correctly. This is possible for the following two reasons. Firstly, PS-algol does not allow a program to access the persistent store directly or to access the implementation of any of its objects. Therefore, the PS-algol type system provides the equivalent of a capability system in software that, with the exception of structure dereferences, is completely static. Secondly, since the CPOMS is part of the PS-algol interpreter, it can only be used with PS-algol programs. Hence, the protection of the persistent store can be provided by the PS-algol compiler.

The combination of the object copying mechanism and the address translation rules has proved particularly efficient. For a program not using the persistent store, there is only a small overhead involved in testing for persistent store addresses. For a program completely contained in the persistent store, the initial performance is poor as all the necessary objects must be imported. However, once these objects have been imported, the performance overhead is minimal. For this class of programs, the additional performance overhead in translating persistent store addresses to heap addresses is estimated to be of the order of 10%[lob87].

4.2 A new architecture

The CPOMS was an experiment in constructing a persistent object store. Based on the experience of the CPOMS, a new architecture was designed. A major influence on the new architecture is the desire to support experiments with concurrency, distribution and transactions as cheaply as possible. As explained earlier, the CPOMS is vertical in structure, making it difficult to alter individual parts without affecting the whole. To overcome this, the new architecture is composed of highly modular horizontal layers. Each layer may be reimplemented independently of the others as long as it preserves its specified interface.

The new architecture distinguishes the use of an object from its storage. One set of layers are provided to implement a persistent object store and another set of layers are provided to implement the desired use of the persistent objects. The independence of the layers allows experiments with the use of an object to be independent of the way the object is stored. As we will see later this is useful in concurrency, transactions and distribution.

4.2.1 Protection

All computer systems attempt to ensure that programs behave correctly. The way in which this is achieved is dependent on the kind of programming language supported. For example, an architecture that supports low level programming languages cannot assume that a program will manipulate its own or indeed other objects correctly.

To overcome this problem, it is necessary to provide a protection mechanism that cannot be circumvented. Since the very nature of low level languages allows them to manipulate the implementation of their objects, a protection mechanism must be provided at the level of the implementation of the objects, that is the store. This may be achieved by allocating programs or individual objects their own address spaces. An object is then protected by controlling the access rights to the object and how the object's address may be distributed. Such a mechanism is known as a capability mechanism and possible implementation techniques are well known. Capability systems were discussed in Section 2.1.

A capability mechanism has several disadvantages. For example, it can only provide protection at a low level. Also, its implementation requires special purpose hardware to operate efficiently and consequently is expensive to modify or reimplement.

High level programming languages do not allow a program to access the implementation of an object or to access an object of the wrong type. For these reasons, a protection mechanism may be employed at the level of the programming language. The effectiveness of this approach has been demonstrated by the PS-algol/ CPOMS system. In such a system, the storage mechanisms are always used correctly, avoiding the need to integrate them with any form of protection mechanism. This has the advantage of allowing the storage mechanisms to be relatively simple and consequently less expensive to modify or reimplement than a capability system. Also, special purpose hardware is not necessary to support an efficient implementation. This was demonstrated, for example, by the efficient software implementation of the object addressing and commit mechanisms of the CPOMS.

The advantages of using a high level programming language can only be realised if all programs are written in that language. This is necessary to ensure that the protection mechanisms used by separate programs are compatible. However, it is not desirable to constrain an architecture to support only one language. For example, some applications may be more conveniently described in an applicative programming language than an imperative one. A solution to this problem, adopted by the new architecture, is to use an intermediate language into which several different programming languages may be compiled. The intention of the intermediate language is to support a range of programming languages that are suitable for most kinds of application. It is not intended to support all possible programming languages since the protection mechanisms used by the languages must be compatible.

The intermediate programming language chosen for the new architecture is PAIL, the **P**ersistent **A**rchitecture **I**ntermediate **L**anguage. It was developed by Dearle[dea87] and

currently supports the imperative programming languages PS-algol and Napier[mor88], and the applicative programming language Hope+[per87].

PAIL describes a high level architecture to which all programs must conform. As such, an implementation of PAIL describes those operations that may be performed on an object and how an object is to be interpreted, but it does not describe how an object is stored. Hence, PAIL can be thought of as a viewing mechanism over the store. Since the separation of an object's use from its storage is an aim of the new architecture, an implementation of PAIL is provided as the central architecture layer.

4.2.2 The abstract machine

To ensure that there is an explicit distinction between the use of an object and its storage, it is necessary to separate PAIL from the store. This is achieved by introducing an *abstract machine* as an architectural layer between the PAIL layer and those layers implementing the store. The abstract machine provides an instruction set suitable for executing PAIL programs. It also provides any primitives required by PAIL to support operations such as concurrency, transactions and distribution. A description of the abstract machine is given elsewhere[bro88a].

The abstract machine is able to access the architecture's store layers. However, it only supports abstract instructions that manipulate persistent objects. Thus, any details concerning the storage of persistent objects are not visible to architecture layers, such as PAIL, that may use the abstract machine. This abstraction over the store, coupled with the provision of all of PAIL's primitives, frees the storage layers from any need to directly support models of concurrency, transactions or distribution. Consequently, the abstract machine guarantees the separation of the use of an object from its storage.

4.2.3 Concurrency, transactions and distribution

The provision of concurrency, distribution and transaction models is restricted to those architectural layers constructed using PAIL, since they deal with logical operations on objects and not how the objects are physically stored. This is true whether the chosen model is explicitly programmed by a user, is a primitive part of a programming language or is a primitive operation of a machine. In fact, these three levels of model correspond exactly to programming in a language that compiles to PAIL, providing the appropriate primitives in PAIL or providing the appropriate primitives in the abstract machine. In each case, it is completely unnecessary to alter the way objects are physically stored.

4.2.4 The layers of the architecture

The new architecture is composed of two sets of layers: a set of layers that implement the use of a persistent object and a set of layers that implement the storage of a persistent object. In the preceding discussion, the architecture layers that manipulate a persistent object have been briefly described. Based on this description, the new architecture's structure can be refined from a two component structure to the following three component structure, as depicted in Figure 4.1:

a) any concurrency, transaction or distribution models that may be constructed from PAIL,
b) PAIL, together with an abstract machine that supports PAIL programs, and
c) a persistent object store.

In the remainder of this chapter, the layers that form the persistent object store are described, followed by a further refinement of the architecture's structure into its individual layers.

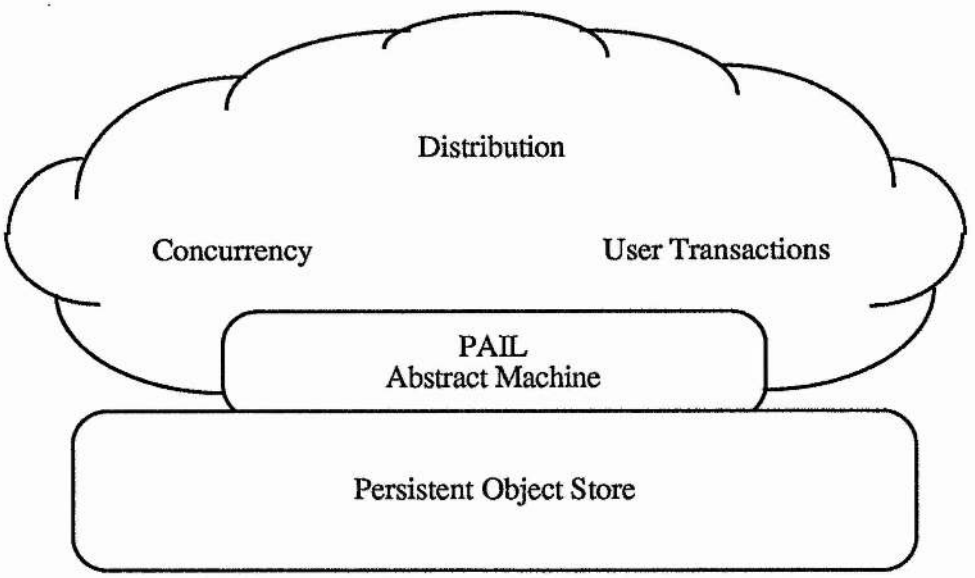


Figure 4.1. The three major architecture components.

4.3 A persistent object store

The persistent object store supported by the new architecture provides the abstract machine with a *heap of persistent objects*, a single object format and a *checkpointing* mechanism. The abstract machine operates on persistent objects without reference to how they are stored. Similarly, the persistent object store is designed to store objects without reference to how they are used. This is achieved by providing a single object format, the interpretation of which is the responsibility of the PAIL layer and the abstract machine. Hence, the persistent object store does not need to support object formats specific to PAIL or the abstract machine, thereby allowing the persistent store to operate independently.

An essential feature of any persistent store is that it is stable. The new architecture's persistent store achieves this by using a checkpointing mechanism. Access to this mechanism is provided to the abstract machine for two reasons. Firstly, it may be made available via PAIL to allow a program to be written that implements a transaction mechanism. Such a program may construct a log of operations to be performed and wish to ensure that the log is in *stable storage* before performing the actual operations. Such a mechanism is described by Krablin[kra85]. Another reason for making the checkpoint

explicit is that it allows the abstract machine to maintain data outwith the persistent store. When a checkpoint is required, the abstract machine can replace any data it keeps in registers or other special hardware, so that the persistent store can operate independently of these devices.

The elimination of support for concurrency and protection allows the store to be a single unshared store, thereby eliminating the need to support different address spaces for programs or individual objects. As a result, the majority of the complex context switching mechanisms present in traditional computer systems can be eliminated. However, the abstract machine layer still needs to support processor scheduling, but without the additional overhead of changing address spaces. That is, the concurrency control mechanisms supported by PAIL and the abstract machine are based on lightweight processes that share the same address space. The potentially high cost of changing address space is one reason for the poor performance of some capability systems that deal with large numbers of objects, see Section 2.1.

Since the store does not need to support distribution, its address space does not need to be larger than the physical store. Thus, the size of an address can be relatively small, even in a very large distributed system. This is in contrast to systems such as Monads, that uses very large addresses to uniquely identify objects across a network of machines. The addressing of objects on remote machines is performed using a *link object* that is maintained by PAIL or the abstract machine. Such link objects may contain the name of the remote machine, together with the object's address on the remote machine. Since an object's address can only be interpreted on its own machine, it is not held in an address field of a link object. This allows different instances of the architecture to pass addresses using different forms.

The checkpointing mechanism used to simulate a stable store is a very simple form of transaction. In fact, it is the minimum level of transaction facility that is required to support stability. In addition to supporting stability, the checkpointing mechanism can also be used

to implement more complex transaction mechanisms. Therefore, PAIL and the abstract machine can be used to implement such transaction mechanisms on top of the persistent store.

4.3.1 The heap of persistent objects

The layer of the persistent store visible to the abstract machine provides a set of object management procedures that may be used by the abstract machine to construct and maintain a heap of persistent objects. Each object in the heap contains a description of its size and the location of any addresses it may contain.

The object management procedures are used to organise a single contiguous stable store that is itself provided as an architecture layer. To ensure that the abstract machine does not need to operate directly on the stable storage, the object management procedures include explicit read and write operations. These procedures allow the abstract machine to read or modify any part of an object. As a direct consequence of this, the exact structure of any particular object and the operations permitted on it are determined solely by the PAIL layer and the abstract machine. That is, the heap of persistent objects is merely a logical structuring of the stable storage. In order for the heap to operate correctly, the following conventions must be adhered to by PAIL and the abstract machine:

a) Objects are only created by the object management procedures.

This convention ensures that all objects are created together with the housekeeping information required by the object management procedures. Furthermore, the stable storage allocated to an object will always be contained within the stable store's address space.

b) Addresses are not manufactured.

This convention ensures that no object can be accessed without following addresses from other objects. That is, a program cannot calculate the address of an object it

would not otherwise be allowed to access. Furthermore, a new address can only be allocated by the creation of a new persistent object.

c) All addresses are held in the address fields of an object.

To support a garbage collection mechanism, it is necessary to be able to identify every address held within the persistent store. This is particularly important if the garbage collector relocates some or all of the persistent objects since every copy of a relocated object's address would need to be identified and modified.

d) All addressing is performed by indexing object addresses.

This convention ensures that a garbage collector will only have to deal with object addresses. That is, the only addresses held in the persistent store are the addresses of objects. This frees a garbage collector from the overheads involved in mapping an arbitrary address to an individual object. Hence, it supports the efficient marking of persistent objects, as well as an efficient implementation of a compacting garbage collector.

e) Reachable objects are not explicitly deleted.

To reduce the frequency with which garbage collections are required, the persistent store provides a mechanism for explicitly deleting objects. However, the delete operation must not be applied to data that may be subsequently accessed since this could result in spurious errors.

The five conventions described above are only effective if they are adhered to by all programs using the architecture. Therefore, to use the total architecture effectively all programming languages should be compiled into PAIL. However, as will be described later, components of the architecture may be used independently to support a wide variety of systems.

An aim of the new architecture is that it will support experimental implementations. As such, each layer of the architecture is required to conform to a specific interface regardless of the implementation techniques it may use. To meet this requirement, an implementation of this layer of the persistent store must provide the following nine object management procedures:

a) initialise object store,
b) close object store,
c) create object,
d) destroy object,
e) first object,
f) read object,
g) write object,
h) checkpoint object store and
i) garbage collect.

4.3.1.1 Initialise object store

This procedure performs those tasks necessary to initialise the heap of persistent objects. As part of this, it causes the stable storage to be initialised. This procedure should be called before the heap of persistent objects is used. It should only be called once.

4.3.1.2 Close object store

This procedure performs those tasks necessary to finish using the heap. In particular, it causes the stable storage to stop in a safe manner. Once this procedure has been called, no further use should be made of the heap of persistent objects or the stable storage.

4.3.1.3 Create object

This procedure is used to create a new persistent object in the heap. It is parameterised by the size of the new object. When the object is created, it is initialised as an object of the required size with no address fields. The act of inserting address fields into the object and

updating the object's description is the responsibility of the abstract machine. Once the object has been created, its address is returned as the result of the procedure. If, for some reason, the object cannot be created, a null address is returned to indicate an error.

4.3.1.4 Destroy object

This procedure is used to destroy a persistent object and release that part of the stable storage allocated to it. It is parameterised by the address of the object to be destroyed. It must never be used on an object that remains accessible.

4.3.1.5 First object

This procedure returns the address of the first persistent object created by the heap. It is this object that is used to determine whether or not an object is accessible. That is, an object is accessible if it can be reached by following object addresses starting from the first object. Hence, the first object is called the *root object* of the persistent store. When an instance of the architecture is first initialised, it creates its own root object. The presence of this object allows the abstract machine to operate without any special knowledge of stable store addresses. In particular, it does not need to know a special address with which it can find the root object.

4.3.1.6 Read object

This procedure is used to copy words from an object in the stable store to physical memory. It is parameterised by the object's address, the offset to the first word to be copied, the number of words to be copied and the physical memory address to which the words are to be copied. The term physical memory is assumed to refer to hardware such as a processor's registers.

4.3.1.7 Write object

This procedure is used to copy words from physical memory to an object in stable storage. It has the same parameters as the *read object* procedure.

4.3.1.8 Checkpoint object store

This procedure makes permanent those changes to the persistent store that occurred since the last checkpoint. To ensure that the persistent store is checkpointed correctly, the abstract machine must ensure that any information kept outwith the store is replaced before this procedure is called. This situation may arise where, for efficiency reasons, information may be temporarily held in a processor's registers or other special purpose hardware. The provision of this procedure transforms the heap of persistent objects into a stable heap.

4.3.1.9 Garbage collect

This procedure identifies and destroys any objects that are not accessible by following addresses from the root object. Like the checkpoint procedure, all information kept outwith the persistent store should be replaced before this procedure is called. Garbage collection will normally be initiated following an unsuccessful attempt to create a new persistent object. If, after a garbage collection, the new object can still not be created, then the heap is full. The new object cannot be created until some existing objects are made inaccessible and successfully garbage collected. How this is achieved is the responsibility of the abstract machine.

To complete the specification of this layer's interface, a single object format must also be specified. This should be designed to suit the experimental environment. For example, if the architecture is to be implemented on a physical architecture such as a SUN workstation[sun86a] or a DEC VAX[dec83], the object format would be based on a 32 bit word. The implementation description given in Chapter 6 is intended for use on such machines and includes a specification of the object format.

4.3.2 Stable storage

The heap of persistent objects is directly supported by a single contiguous stable store. In turn, the stable store is supported by a single contiguous *non-volatile store*. The purpose of

the stable store layer is to provide access to a contiguous section of non-volatile store that is always in a self-consistent state. This is achieved by two complimentary mechanisms.

The first mechanism allows all of the architecture's non-volatile store to be directly addressed as if it were a contiguous virtual memory. To support this, an address mapping is required from non-volatile store to the virtual memory. Since there is only one store supported by the architecture, addresses in the non-volatile store can be mapped directly to addresses in the virtual memory. Thus, an object can have the same address in the non-volatile store as it does in the virtual memory. The use of a direct mapping removes the need for the mapping tables used by traditional virtual addressing mechanisms.

The second mechanism is used to maintain a section of the virtual memory in a self-consistent state. The non-volatile store is assumed to be organised as a contiguous set of equal sized blocks. When a block of non-volatile store is to be accessed it is first copied into main memory. The copy of the block may then be read or written as required.

The combination of the two mechanisms corresponds to the paging mechanism of a traditional virtual memory with one significant difference. In a conventional paging scheme, a modified block is simply written back to its original location on non-volatile store. However, to maintain a section of the non-volatile store in a self-consistent state this must be performed by a transaction mechanism. The transaction mechanism operates by using the blocks of non-volatile store, outwith the self-consistent section, to maintain copies of modified blocks. These copies are then available to reconstruct the original state of the self-consistent section if a failure occurs. When all the currently modified blocks have been written to non-volatile store, a new self-consistent state can be achieved by ignoring the original copies of the modified blocks. This combination of a transaction mechanism and a paging scheme is known as shadow paging, see Section 2.4. Hence the stable storage layer can be described as a shadow paged virtual memory.

An implementation of this layer of the persistent store must provide the following seven interface procedures:

a) initialise stable store,
b) close stable store,
c) read words,
d) write words,
e) checkpoint stable store,
f) data start and
g) data size.

4.3.2.1 Initialise stable store

This procedure performs those operations necessary to initialise the stable storage. This should be the first procedure that is called. It should only be called once.

4.3.2.2 Close stable store

This procedure is used to stop the stable storage mechanism. As part of this, it ensures that the stable store is in a self-consistent state. Once this procedure has been called, no further use should be made of the stable store.

4.3.2.3 Read words

This procedure is used to copy words from the virtual memory to physical memory. It is parameterised by the virtual address of the words to be copied, the number of words to be copied and the physical memory address the words are to be copied to. The term physical memory is assumed to refer to hardware such as a processor's registers.

4.3.2.4 Write words

This procedure is used to copy words from physical memory to the virtual memory. It has the same parameters as the *read words* procedure. However, as part of the copying it should mark those pages of virtual memory that it modifies.

4.3.2.5 Checkpoint stable store

This procedure is used to ensure that all changes to pages since the last call of this procedure are either made permanent or are all undone. In particular, it must ensure that the original contents of a page in the stable storage are copied, before it is overwritten by the modified contents. This allows any changes to be undone if a failure prevents the checkpoint from completing.

4.3.2.6 Data start

This procedure returns the lowest address of the virtual address space that may be used by the persistent heap layer. This address is used to calculate the address of the persistent heap's root object.

4.3.2.7 Data size

This procedure returns the length of the virtual address space that may be used. In conjunction with *data start*, this procedure describes the valid range of virtual addresses that may be used.

4.3.3 Non-volatile storage

The stable store layer requires the provision of a non-volatile store that is assumed to consist of a contiguous set of equal sized blocks. Subject to these constraints, the non-volatile store may actually consist of as many disjoint physical components as desired. To meet the specification for this layer of the persistent store, an implementation of non-volatile storage must provide the following six procedures:

a) initialise storage,

b) close storage,

c) read block,

d) write block,

e) block size and

f) synchronise.

4.3.3.1 Initialise storage

This procedure performs those operations necessary to initialise the non-volatile storage. For example, it may need to gain exclusive use of the storage media implementing the store. This should be called before the non-volatile store is used. It should only be called once.

4.3.3.2 Close storage

This procedure performs those operations necessary to stop using the non-volatile store. This may include relinquishing exclusive use of the storage media. This should be the last procedure to be called.

4.3.3.3 Read block

This procedure is used to copy a block of non-volatile storage to a page of main memory. It is parameterised by the block number in the non-volatile store and the page number in main memory.

4.3.3.4 Write block

This procedure is used to copy a page of main memory to a block of the non-volatile store. It has the same parameters as the *read block* procedure.

4.3.3.5 Block size

This procedure returns the size of a block of non-volatile store.

4.3.3.6 Synchronise

This procedure does not return until all requested *read block* and *write block* operations have been completed. This ensures that a checkpoint operation knows that a requested operation has completed.

To complement these six procedures the following assumptions must be valid:

a) A *write block* can only corrupt the block being written to.
b) If a *write block* completes it has not caused any corruption.
c) A block is written in a predefined order, low addresses to high addresses or high addresses to low addresses.
d) A *read block* can never cause corruption.
e) The architecture halts if a *read block* or *write block* fails.

These procedures and assumptions are sufficient to implement the stable storage mechanism, described in Section 4.3.2, for the following reasons. Firstly, all completed *read block* and *write block* operations performed by the checkpoint mechanism can be assumed to have worked. The checkpoint mechanism can ensure all requested reads and writes are complete by using the *synchronise* procedure. Since the failure of a *read block* or *write block* causes the architecture to halt, failures can be ignored during normal operation of the store. In fact, failures need only be detected when the architecture is restarted. This can be performed by updating certain blocks at each step in a checkpoint. Each of these blocks have a time stamp recorded at their start and end. Since all blocks are written in a specific order, a corrupt block can be detected by the discrepancy between its two time stamp values. Thus, the point of failure can be determined by identifying the corrupt block. The appropriate action can then be taken to restore the stable store to a self-consistent state.

4.4 The architecture layers

The new architecture is composed of the following eight layers shown in Figure 4.2:

a) A physical machine that provides main memory, non-volatile storage and network interfaces.

b) The lowest software layer is a set of interfaces to allow access to main memory, the non-volatile storage and the appropriate network communications software.

c) A fixed size non-volatile store that is organised as a contiguous set of equal sized blocks. It may be constructed from an arbitrary number of disjoint physical components but must appear to be a single contiguous set of blocks.

d) A shadow paged virtual memory mechanism over the non-volatile store with a simple checkpointing mechanism. The checkpointing mechanism operates as a secure atomic action. The size of the virtual address space is the same as the physical size of the non-volatile store. However, only part of the virtual address space is shadow paged, the remainder is used to hold the shadow copies of changed blocks.

e) A set of object management procedures that can be used to implement a heap of persistent objects within the virtual address space. These procedures assume that all objects conform to a single format that distinguishes an object's address fields. In addition, the following assumptions are made of the higher layers of the architecture:

   a. they do not create their own objects,
   b. they do not manufacture their own addresses,
   c. they store all addresses in the address fields of objects,
   d. they perform all addressing by indexing object addresses and
   e. they do not explicitly delete reachable objects.

f) An abstract machine that provides an instruction set and those primitives required to efficiently execute PAIL programs.

g) Any programming language that can be compiled into the language PAIL. The use of PAIL ensures that all programs conform to the assumptions of layer **e**.

h) Any concurrency, distribution or user transaction models that are present in or can be constructed from the programming languages of the previous layer.

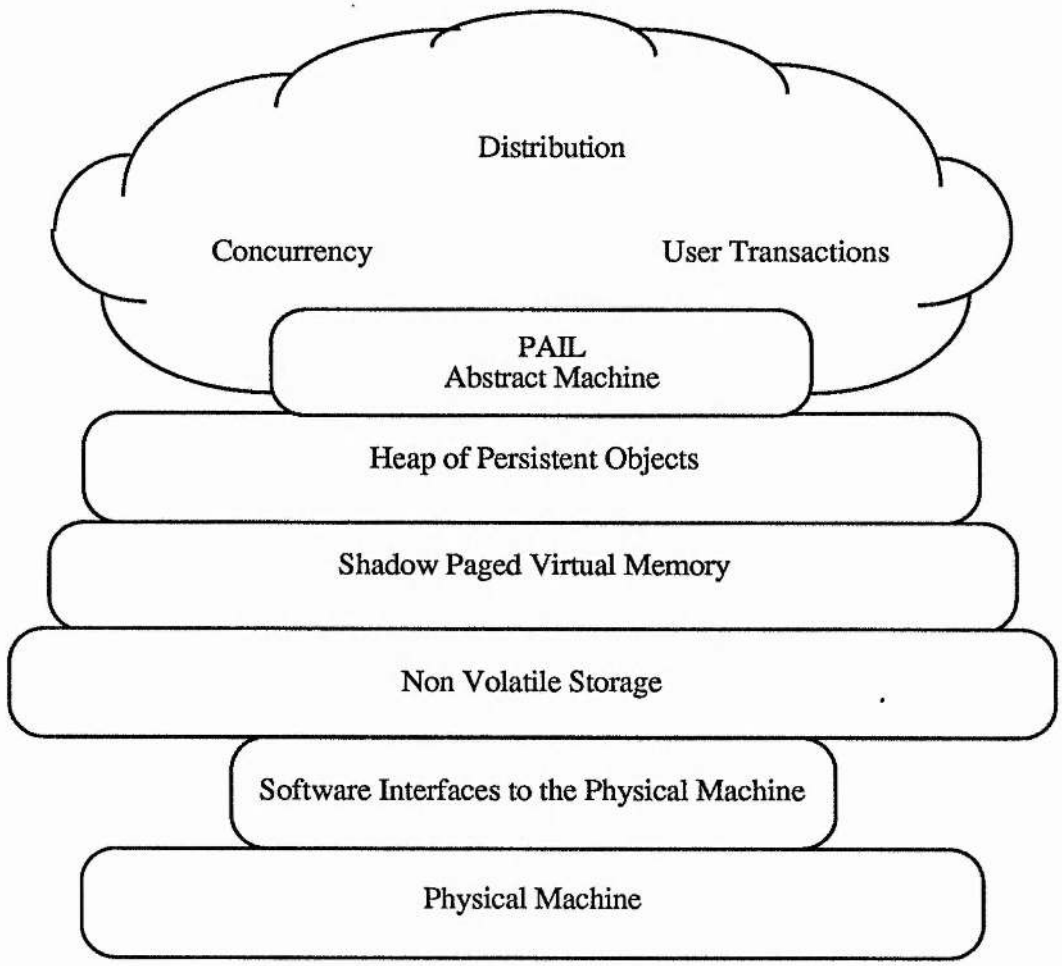


Figure 4.2. The complete layered architecture.

4.5 Conclusion

The new architecture is composed of several distinct layers. To support experimentation, each layer may be reimplemented independently of the other layers provided that it preserves its specified interface. This permits new versions of the architecture to be rapidly and economically produced. Furthermore, the individual components of the architecture may be made available for use in totally different architectures. For example, in the first UNIX implementation of the architecture, the layers that provide the heap of persistent objects are available as a C library[ker78]. Therefore, they could be used to provide a persistent heap for any application developed in a UNIX/ C environment. However, such a persistent heap would not be allowed to interact with an instance of the total architecture since the protection mechanisms employed may not be compatible.

The particular choice of architecture layers distinguish the basic mechanisms required to support a persistent system such as stable storage, persistence, concurrency control, distribution and transactions. The simplicity of the resulting architecture and its potential for experimentation are in direct contrast to the complex and inflexible nature of traditional computer architectures.

5 Implementing the layered architecture

The remainder of this thesis describes an implementation of the layered architecture described in the previous chapter. Since the thesis is concerned with persistent stores, the description is confined to the layers that support the heap of persistent objects and how these layers are interfaced to the abstract machine. Implementations of the other layers of the architecture are described elsewhere, a description of PAIL is given by Dearle[dea87], a model for concurrency control is given by Morrison[mor87] and a model for transactions is given by Krablin[kra85]. The support provided for distribution is described in Chapter 7 together with a possible distribution mechanism.

5.1 Support for virtual address translation

The layers that implement the heap of persistent objects operate within a virtual address space that is mapped onto the non-volatile storage. The efficiency of the virtual addressing mechanism is therefore a major factor in the overall performance of the architecture.

The ideal implementation of the virtual addressing mechanism would be based on special purpose hardware similar to that used on the Atlas machine, see Section 2.2. Since the virtual addressing mechanism does not need to support multiple address spaces, the mapping from virtual addresses to addresses in non-volatile store can be one to one. Therefore, there is no need to support an explicit mapping between virtual addresses and non-volatile storage. A consequence of this is that to support a paging mechanism, it is only necessary to record mappings for those pages present in main memory since there is an implicit address mapping for every other page in the virtual address space.

In the Atlas machine, the pages present in main memory were recorded within an associative store that contained an entry for every physical page of main memory. A virtual address can then be mapped to a physical address in one machine cycle, allowing the virtual addressing mechanism to be both simple and efficient. The performance of this addressing mechanism

may be further enhanced if the hardware implementing the associative store is given additional functionality.

In contrast to the performance advantages, a hardware based addressing mechanism has several major disadvantages when used for experimentation. These disadvantages are related to the cost and complexity of constructing special purpose hardware, which can be both time consuming and expensive. The associative store described above, with further enhancements, may be quite complex. This complexity would be further increased if the associative store is extended as the available main memory is extended. The complexity increases the potential for errors in the design, thereby increasing the time required to construct the hardware and further increasing the cost.

In a research environment, the use of special purpose hardware may prove too costly to develop and maintain. Such hardware is difficult to modify once it has been constructed and can only be used by a small community. Also, due to the high cost of construction, it may be difficult to adopt new hardware technologies as they become available.

Many of the disadvantages of using special purpose hardware can be avoided by adopting a software address translation mechanism. In comparison with the costs of designing and developing hardware, a software mechanism is relatively cheap and relatively easy to modify. Some other advantages of a software solution include the ease with which software may be ported to new hardware and the potentially large community to which it is then available.

Since one aim of the layered architecture is to support experimentation, there is considerable advantage to be gained by adopting a software address translation mechanism. However, in comparison with the hardware translation mechanism described above, any software mechanism will be many times slower. This could significantly degrade the overall performance of the architecture. The effect may be partially overcome if the number of

address translations required are minimised. An example of how this may be achieved and its effectiveness is demonstrated by the CPOMS, described in Chapter 3.

5.2 Extensions to the architecture layers

The layered architecture has been extended to utilise the techniques employed by the CPOMS, that minimise the number of address translations required and therefore improve the overall performance. The extension involves the addition of an extra layer between the abstract machine and the heap of persistent objects, as shown in Figure 5.1. The additional layer is a heap of objects that operates solely within main memory, with all of the addressing being performed with main memory addresses. The operation of the new layer corresponds exactly to the heap of a PS-algol program. Similarly, the heap of persistent objects corresponds to the databases supported by the CPOMS. Hence, the object addressing and commit mechanisms of the CPOMS can be adapted to manage the movement of objects between the new layer and the heap of persistent objects.

The basic difference between the CPOMS and the layered architecture is that the store does not have to support concurrency and transactions. That is, there is no need for the complexities introduced by the PS-algol databases, thereby eliminating a substantial part of the CPOMS. Thus, the CPOMS mechanisms, adapted for use within the layered architecture, need only support the movement of objects between the new layer and the heap of persistent objects.

To ensure that the architecture layers retain their independence, the new layer has its own interface definition allowing it to be reimplemented independently of the other layers. The interface to the new layer is similar to that of the heap of persistent objects but it differs in the following four ways:

a) there are no explicit read and write procedures,

b) an interface procedure is provided to garbage collect the heap of persistent objects,

c) an explicit address translation procedure is provided and

d) a convention is specified to differentiate stable store addresses and main memory addresses.

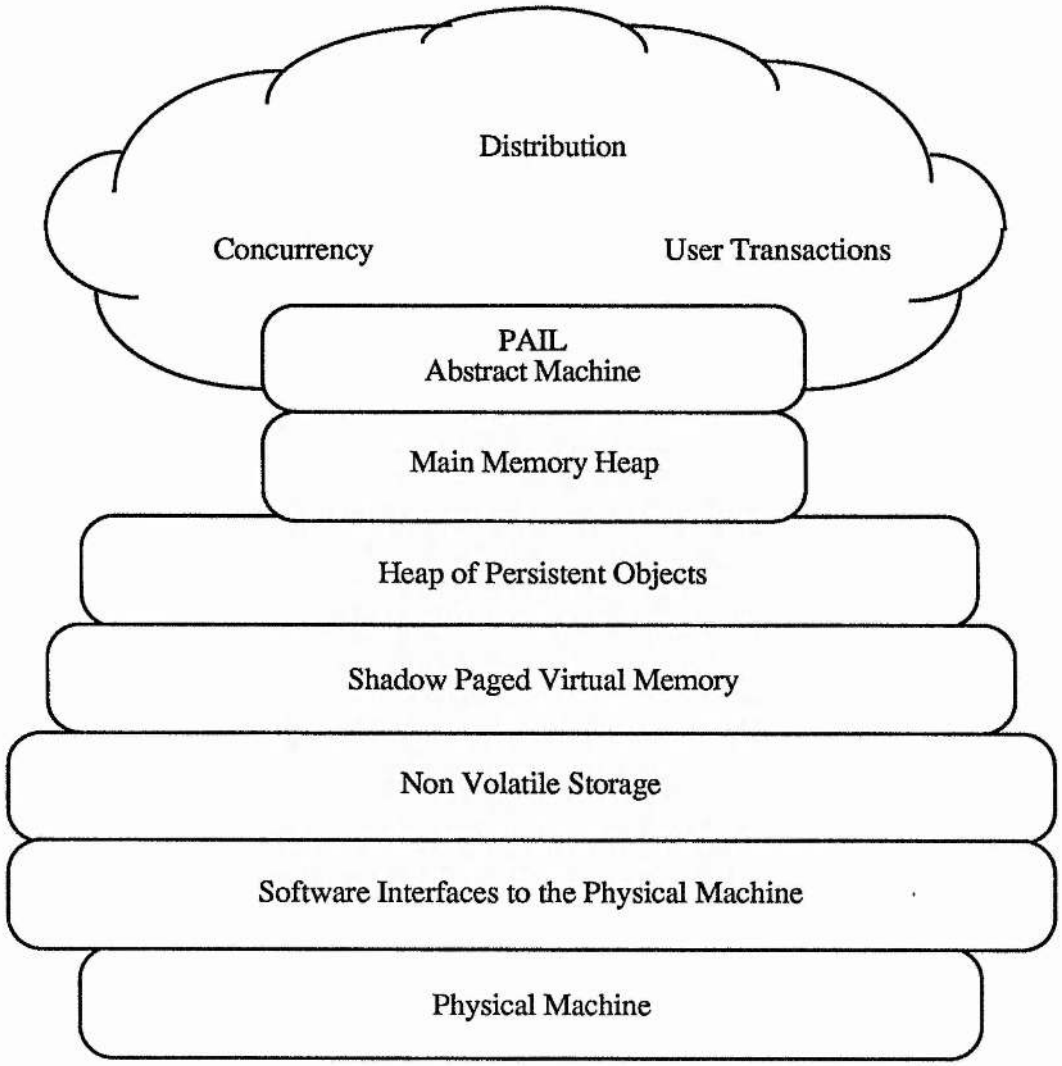


Figure 5.1. The revised layered architecture.

5.2.1 Read and write procedures

Since the abstract machine operates directly on the main memory, there are no explicit read and write procedures. However, this does constrain the addresses of objects in the new layer to be main memory addresses.

5.2.2 Garbage collection

Another difference between the interfaces is the provision of an interface procedure to invoke a garbage collection within the heap of persistent objects. This procedure allows the garbage collection to be explicitly invoked during periods of low activity, thereby avoiding potentially long delays at inconvenient moments. To ensure that the garbage collection operates correctly, the current state of the main memory heap must be propagated to the heap of persistent objects. This may be achieved by first invoking the *checkpoint* procedure for the main memory heap.

5.2.3 Explicit address translation

The third difference between the interfaces is the addition of an explicit address translation procedure. As in the PS-algol/ CPOMS system, this procedure is used by the abstract machine to explicitly translate stable store addresses, when a main memory address is required. It is envisaged that the address translation rules of the PS-algol/ CPOMS system will be adapted to suit the particular abstract machine used by the architecture.

5.2.4 Identifying stable store addresses

To support the use of the address translation procedure, the interface also describes how a stable store address may be differentiated from a main memory address. The convention adopted is to record stable store addresses as illegal main memory addresses. For example, when an object is copied from the heap of persistent objects each of its address fields may be negated, thereby transforming their contents into illegal main memory addresses.

5.3 Conclusion

The extensions to the architecture, described above, preserve the independence of the layers with just one exception. The abstract machine implementation must be tailored to operate either with a heap in main memory or a heap in stable storage. However, the implementation of an abstract machine, whether by a code generator or an interpreter, is always hardware dependent so this one loss of layer independence was considered acceptable. In either case, the potential of the architecture for experimentation is not significantly affected.

6 A software implementation of the architecture

This chapter describes the first implementation of the persistent object store supported by the layered architecture. The description is divided into five parts:

a) the relevant features of the abstract machine,

b) the main memory heap,

c) the heap of persistent objects,

d) the stable storage, and

e) the non-volatile storage.

6.1 The persistent abstract machine

The abstract machine is designed to support the execution of PAIL programs. PAIL can support a wide range of block structured programming languages some of which may provide first class procedures, polymorphism and abstract data types. An example of a programming language that provides all three is Napier[mor88]. Hence, the abstract machine must be able to support first class procedures, polymorphism and abstract data types.

A consequence of supporting first class procedures is that variables declared within a block may be retained beyond the normal life time of the block, as part of a procedure's static environment. In a traditional computer architecture, the storage associated with a block is allocated from a stack and automatically recovered when the code for the block has been executed, releasing the storage associated the variables declared within the block. To support languages with first class procedures, the abstract machine implements the block retention mechanism used by the PS-algol abstract machine[psa85].

The block retention scheme, supported by the abstract machine, is based on the creation of a separate persistent object for each block. The object contains the stack associated with the

block's execution and is known as a frame. To complement the use of separate stack objects, all executable code is also in the form of separate persistent objects, one per procedure. An instance of a procedure is represented by two object addresses, one addressing the procedure's code object and one addressing the procedure's static link. This is known as the closure of the procedure.

When a procedure is called or a block is executed a new frame is created. The frame's dynamic link is the address of the caller's frame. Hence, all frames that form the dynamic chain are reachable from the frame of the executing procedure or block. This frame is known as the current frame. Similarly, the frames that form the static environment of a procedure are reachable via the procedure's static link. When a procedure or block terminates, the associated frame is no longer part of the dynamic chain. Unless the frame is to be retained as part of a procedure's static environment, it is no longer reachable from the program and may be garbage collected. Thus, the garbage collector can automatically destroy or retain a frame as appropriate.

A description of the implementation techniques that support polymorphism and abstract data types is given by Dearle[dea88]. They are not described here since they are not relevant to the implementation of the persistent object store.

In addition to supporting PAIL, the abstract machine is constrained to use the persistent object store as its only available storage. This requires all the storage used by the abstract machine to be mapped onto persistent objects and it prohibits the use of known addresses to access predefined objects such as error handling procedures or literal values.

To support the use of predefined objects, the abstract machine maintains a special object that is large enough to contain the address of every predefined object that may be required. When the architecture is first used, the abstract machine creates the special object and places its address in the root object of the heap of persistent objects. Thereafter, the special object

can be accessed via the root object, whose address is provided by the interface to the heap of persistent objects. The predefined objects can then be accessed via the special object.

The combination of the block retention scheme and the persistent object store has an important consequence for the operation of the abstract machine; in that there is no separation of the available storage between a program's data and a program's dynamic state. In a traditional computer architecture, this division of storage may be in the form of a heap and a stack. When either the heap or stack is exhausted a program will fail even if there is storage available in the other. The advantage of eliminating the division is that a program will only fail when the entire storage has been exhausted.

The other features of the abstract machine are not relevant to the implementation of the architecture's persistent store and are described elsewhere[bro88a].

6.2 The main memory heap

The main memory heap of the layered architecture corresponds exactly to the heap used by the PS-algol interpreter. The purpose of the heap is to provide a cache of persistent objects. This is achieved by providing a simplified form of the CPOMS to manage the movement of objects between the heap of persistent objects and the main memory heap. The first implementation of the main memory heap operates as follows.

6.2.1 The heap organisation

The heap is divided into two parts, a contiguous collection of objects and a distributed mapping table. The storage allocated to the objects starts from the low address end of the heap and grows towards high addresses as objects are created. When an object is created an additional word is allocated that prefixes the new object. The extra word forms part of the mapping table and not part of the object. In addition to the prefixed words, the mapping table also consists of a separate storage area. This area starts at the high address end of the heap and grows towards low addresses, as shown in Figure 6.1. The mapping table is also

extended when an object is created to ensure that the mapping table can record a mapping for every heap object, see Section 6.2.3.

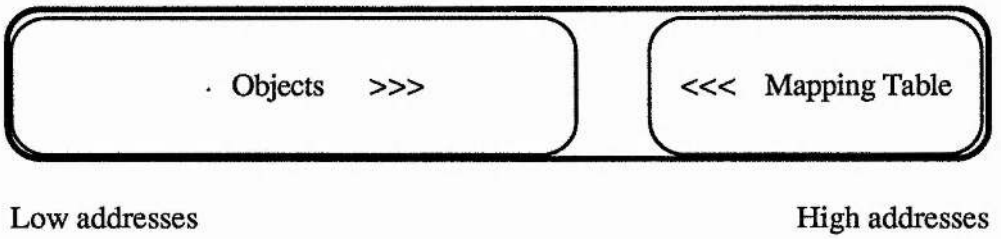


Figure 6.1. The layout of the main memory heap.

When the extension of either storage area would cause them to interfere with each other a garbage collection is invoked. The purpose of the garbage collection is to compact the reachable objects towards the start of the heap and allow the object storage to be contracted. Similarly, a reduction in the total number of heap objects also allows the mapping table to be contracted.

6.2.2 The format of persistent objects

In the first implementation of the layered architecture, all persistent objects have the format illustrated in Figure 6.2. The first word contains a count of the number of address fields within the object and the second word records the total size of the object in words. A word is a 32 bit integer. To support garbage collectors and other storage utilities, the first word reserves its 8 most significant bits for use as markers. The remaining 24 bits hold the number of address fields. Within the local heap the marker bits are used to:

a) identify objects that must be retained by a garbage collector,
b) identify objects that have been changed,
c) identify objects that may not conform to the address translation rules and
d) distinguish an object's first word from a main memory address, this is discussed in Appendix 1.

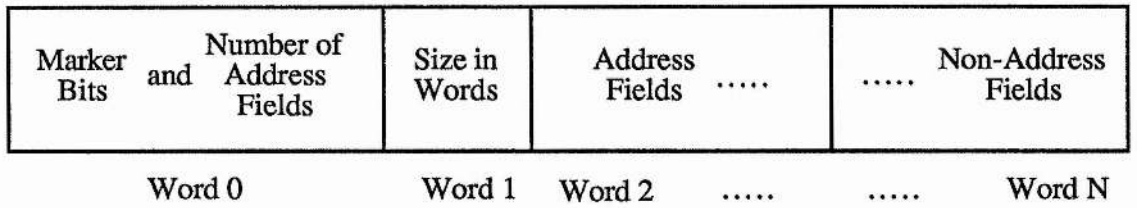


Figure 6.2. A persistent object.

6.2.3 Address translation

The mapping table is used to map persistent object addresses, known as keys, to main memory addresses. The mapping table is known as the key to RAM address table, KRT, since the main memory is assumed to be Random Access Memory. The KRT is functionally equivalent to the PIDLAM used by the CPOMS, see Chapter 3. As such, it provides mappings from keys to main memory addresses to support the copying of persistent objects and it provides mappings from main memory addresses to keys to support writing copied objects back to the persistent store. The KRT provides both mappings as follows.

Firstly, each entry in the KRT is in the form of a main memory address and a key. The key is held in the word that prefixes the copy of the object in main memory and the main memory address is held in the KRT's storage area at the top of the heap. Thus, the mapping from an object's main memory address to its key is performed by indexing its main memory address, this is illustrated in Figure 6.3.

Within the KRT's storage area, the main memory addresses are held in a contiguous set of words starting from the high address end of the storage. The addresses are ordered by their corresponding keys and are searched using a binary split algorithm. Since the addresses and keys are not held as pairs, the corresponding keys must be found by indexing the addresses. Thus, a mapping from a key to a main memory address may involve performing several mappings from addresses to keys as the KRT is searched.

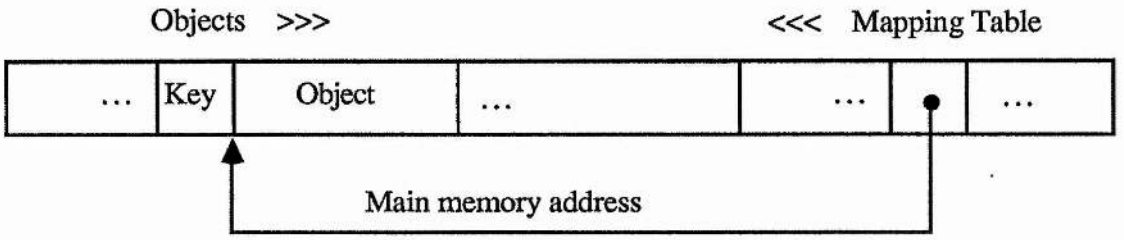


Figure 6.3. The structure of the mapping table.

To ensure that the KRT can record a mapping for every heap object, the KRT's storage area is allocated a single word for every object in the heap. Consequently, the KRT's storage area contains a single word for every heap object without a key. These words form a contiguous area that is made available to the heap's garbage collector to support a non recursive marking algorithm, this is described in Appendix 1.

When a persistent object is first used it is only accessible via its key. The abstract machine is unable to interpret keys, so the key is translated into a main memory address. To allow the abstract machine to distinguish keys, they are held as negative values whereas main memory addresses are positive values. When a key is encountered, the abstract machine explicitly invokes the illegal address procedure provided by the interface to the heap.

The translation takes the form of copying the persistent object into the main memory heap, recording the mapping from the object's key to the copy's main memory address in the KRT and then returning the copy's address. Thereafter, all operations on the persistent object will be performed on the copy. A future translation of the key can lookup the KRT to find the copy's address without the overhead of making an additional copy.

6.2.4 Address translation rules

The main memory heap is implemented purely by software. Therefore, the cost of checking for keys and translating them to main memory addresses can be significant. In order to minimise the potential cost, the main memory heap employs a set of address translation rules

that have the following three effects. Firstly, they minimise the number of times a key must be translated to a main memory address. Secondly, the rules minimise the number of checks necessary to identify addresses that must be main memory addresses. Finally, by ensuring that certain addresses are main memory addresses the objects they refer to must be present in the main memory heap.

The rules were designed to take advantage of the stack oriented nature of the abstract machine and to permit the main memory heap to purge as many objects as possible. Object purging is necessary to support programs that may access or create more objects than the main memory can hold. When the main memory becomes full, a checkpoint is performed and sufficient objects are purged to allow a program to continue, this is described in Section 6.2.9. This operation may fail if the main memory becomes full of objects whose presence is required by the address translation rules. In order to avoid this problem, the rules are designed to minimise the number of objects that must be present.

The complete set of rules are as follows:

a) When a key is translated to a main memory address, the location that held the key is overwritten by the main memory address. The effect of this rule is to minimise the number of address translations since any future use of the location will yield a main memory address.

b) When an address is loaded onto the stack of the current frame it must be a main memory address. Hence, an attempt to load a key onto the stack will result in the key first being translated. This rule has the effect of minimising the number of checks for keys since all the addresses on the current frame's stack are assumed to be main memory addresses.

c) Before an address can be dereferenced it must be a main memory address. This requires all addresses to be checked before they are used unless they are held on the current frame's stack.

d) The address fields of the current frame must contain main memory addresses with the exception of the dynamic link, static link and the display. Consequently, when the dynamic link, static link or display are used, a check must be made to ensure that the address to be dereferenced is not a key. This rule must be enforced whenever a frame becomes the current frame. For example, when a procedure returns to its caller, the caller's frame becomes the current frame and the rule must then be applied. The effect of this rule is to ensure that the objects directly accessed from the stack of the current frame are always present in main memory.

e) When an object is copied into the main memory heap it is marked to indicate that it may not conform to the address translation rules. Similarly, when an address field of an object is overwritten by a key the object is marked. This object marking can be used to determine if a frame conforms to the previous rule. For example, on a procedure return a check is made to see if the caller's frame has been marked. If it has then each of its address fields must be checked to ensure that they meet the requirements of the previous rule. This allows some of the frames on the dynamic call chain to be temporarily removed from the heap without the need to introduce expensive checking on a procedure exit.

f) The root object of the main memory heap and those housekeeping objects addressed by the root object, only contain main memory addresses. This rule ensures that any objects required to support the management of the main memory heap are always present. The combination of rules **d** and **f** define the minimum number of objects that must be present.

6.2.5 Garbage collection

The garbage collection algorithm used by the main memory heap is an adaption of a compaction algorithm given by Morris[mor78]. The purpose of the algorithm is to slide all the accessible objects towards the bottom of the heap, as shown in Figure 6.1, thereby allowing all storage allocation to be performed without any fragmentation problems. That is, all the heap's free space is available as a single unit. A full description of the garbage collection algorithm is given in Appendix 1.

6.2.6 Checkpointing the heap

The interface to the main memory heap provides a checkpoint mechanism to implement stability. When it is invoked, all new or changed objects are copied from the main memory heap to the heap of persistent objects and then the persistent store's own checkpoint mechanism is invoked.

Since the heap's root object persists, all the reachable objects in the heap must also persist. In addition to determining which objects are reachable, the heap's root object is also used to record the abstract machine's dynamic state. Thus, a checkpoint will preserve the abstract machine's dynamic state. A consequence of this is that, if the persistent store fails, it will be restored to a self-consistent state that allows an active program to continue as if nothing had happened. In certain circumstances, this may lead to the same error that caused the persistent store to fail. However, the potential difficulties can be avoided by permitting some special actions to be taken when the abstract machine is restarted.

In contrast to the disadvantages of saving the dynamic state there are considerable advantages to be gained. For example, if a program is simulating a transaction mechanism it need only save a record of its changes since the dynamic state of the mechanism can be restored. Another advantage is that a programmer need not be aware of when a checkpoint occurs since his program can be automatically restarted and continued as if nothing had happened. This is particularly important for a programmer who is writing a series of

programs that operate concurrently on shared data. Consequently, an implementation of the architecture could automatically invoke checkpoints without affecting a programmer's ability to reason about his programs or the system.

6.2.7 The checkpoint algorithm

The algorithm used to checkpoint the main memory heap is performed in the following four steps:

a) marking the reachable objects,
b) allocating keys to new objects,
c) copying all new or changed objects to the heap of persistent objects and
d) checkpointing the heap of persistent objects.

6.2.7.1 Marking the reachable objects

The checkpoint must identify all the objects that it will operate on. These are exactly the same objects that a garbage collection would retain. Hence, the first step of the checkpoint is to invoke the garbage collector's marking algorithm, described in Appendix 1.

6.2.7.2 Allocating keys to new objects

Within the heap of persistent objects, main memory addresses cannot be interpreted. Thus, it is necessary to ensure that all objects have a key by which they may be identified. This is achieved by scanning the main memory heap and allocating a key to each marked object that does not have one. Only marked objects are given keys since unmarked objects are unreachable and will be garbage collected. During this scan the markers left by the garbage collector are removed.

The allocation of a key to an object is performed by creating an object of the same size in the heap of persistent objects. A mapping from the resulting key to the object's main memory address is then entered into the KRT.

6.2.7.3 Copying objects to the heap of persistent objects

When every reachable object has been given a key, all the new or changed objects are copied to the heap of persistent objects. Unchanged objects are not copied because their copy in the persistent store is the same. This step is performed by scanning the heap and copying objects that are marked as changed. As each object is copied, the changed marker is reset to indicate that the object has the same value in both heaps. The change markers are set whenever an abstract machine instruction modifies an object. Similarly, all new objects are marked as changed when they are created.

6.2.7.4 Checkpointing the heap of persistent objects

The final step of the checkpoint is to invoke the checkpoint mechanism provided by the heap of persistent objects. This ensures that the heap of persistent objects has propagated all the changes to non-volatile store.

6.2.8 Garbage collecting the heap of persistent objects

The interface to the main memory heap provides a procedure that can be used to cause a garbage collection within the heap of persistent objects. The garbage collector can only operate correctly if the heap of persistent objects is up to date. Thus, the implementation of this interface procedure must first invoke the checkpointing mechanism described above and only then invoke the *garbage collect* interface procedure provided by the heap of persistent objects.

When the heap of persistent objects has been garbage collected, the keys allocated to persistent objects may be different. For example, if the keys directly address objects in the stable store then the relocation of an object will change its key. As a result, the keys held in the main memory heap may no longer refer to the same persistent objects.

This problem may be overcome by discarding every object in the main memory heap, discarding all the mappings in the KRT and then restarting the abstract machine. Since the

garbage collection followed immediately after a checkpoint, no data will be lost. The only visible effect on the operation of the system will be a delay while the abstract machine is restarted and a subsequent loss of performance as the objects being used are retrieved from the persistent store.

6.2.9 Discarding objects

When an object is accessed, it is first copied into the main memory heap. As a consequence of this, the maximum number of objects that can be accessed at any one time is limited by the size of the heap. To support programs that may access and possibly modify the entire persistent store, it is necessary to overcome this restriction. This can be achieved by discarding objects from the heap while adhering to the address translation rules.

An object should only be discarded from the main memory heap if it has a copy with the same value in the persistent store. Thus, a changed object must be copied to the persistent store before it can be discarded. Since the checkpoint algorithm already exists to perform this task it was extended to automatically discard objects. The modifications to the checkpoint algorithm are as follows.

In step **b** of the algorithm, the markers used by the garbage collector are not removed. Hence, at the end of step **b** all the reachable objects in the heap have a key and are still marked.

The address translation rules describe those objects that must be present in the main memory heap. This information is used to clear the garbage collector's markers from those objects that must be retained. Thus, the checkpoint uses the markers to identify those objects that may be discarded.

The next step in the modified algorithm is to inspect the address fields of every object in the heap. Each field that contains the main memory address of a marked object is overwritten by

the object's key. Thus, at the end of this step of the algorithm the only copy of a marked object's main memory address is held in the KRT.

The remainder of the checkpoint algorithm is unchanged. However, it should be noted that no part of the above algorithm actually discards an object. Furthermore, the KRT still contains a mapping for the discarded objects preventing them being deleted by the garbage collector. To complete the purging algorithm both the garbage collector's mark algorithm and the illegal address procedure are modified.

At the end of the modified checkpoint algorithm, the discarded objects are still marked. Furthermore, they are the only marked objects in the heap. This information is used by the garbage collector to force these objects to be deleted. However, between a checkpoint and the next garbage collection an object's key may be translated into a main memory address. In such a case the object should not be discarded. To support this the illegal address procedure unmarks any object whose key it translates. This transforms the object purging algorithm into a second chance algorithm[pet83].

To complete the object purging, the garbage collector's mark algorithm must be modified. When the mark algorithm initially scans the KRT it marks unmarked objects and unmarks marked objects. In addition it also removes the KRT entries for the previously marked objects. Since these objects were only reachable from the KRT, they will become both unreachable and unmarked. The next garbage collection will then delete them.

6.2.10 Summary

The main memory heap is a simplified form of the CPOMS that is free from the complexities of concurrency constraints and complex transaction mechanisms. Furthermore, it provides an efficient implementation strategy for the architecture's persistent store that does not require special purpose hardware.

6.3 The heap of persistent objects

An implementation of the heap of persistent objects must address several issues. These include:

a) How much of the stable store can be modified between checkpoints?

b) How long will a garbage collection take?

c) Can the system be stopped for the length of time required for a garbage collection?

d) Which garbage collection algorithm is best suited to the size of the heap?

e) Should the heap support direct or indirect object addressing?

The amount of data that can be modified between checkpoints of the stable store is available as part of the stable store's interface, this is described in Chapter 8. However, this information is specific to an instance of the architecture and may not be available to an implementor of the heap of persistent objects. Therefore, an implementation of a garbage collector must assume that several checkpoints may occur during each garbage collection. Furthermore, it must be able to restart the garbage collection after any of the checkpoints.

Another major issue that must be addressed in implementing the heap of persistent objects is the potential size of the heap. This has several consequences for the overall efficiency of the architecture. For example, the garbage collection algorithm used by the main memory heap may perform acceptably for a small heap but prove too slow for a very large heap. This problem would be further aggravated if the system to be supported could not be stopped for more than a few seconds at a time. Such a system would require a garbage collector that could operate concurrently with a user's programs.

The size of the heap also affects the choice of direct or indirect object addressing. For example, a very large heap that supported direct addressing would be unable to relocate objects during a garbage collection, due to the high cost of identifying and modifying all the

copies of an object's address. Alternatively, an implementation of a small heap may not wish to support indirect addressing due to the space overhead involved.

The first implementation of the heap of persistent objects was designed to support a heap of a few megabytes in size. As such, the garbage collection mechanism was chosen for simplicity without regard to efficiency.

6.3.1 The heap organisation

The heap is organised as two equal sized sections often known as semispaces[fen69]. The low address end of each section contains objects and grows towards high addresses whereas the high address end of each section contains one word for each object and grows towards low addresses. The high address end of the second section is used to hold the stable store address for each object. It is known as the indirection table. The high address end of the other section is unused. The organisation of the heap is shown in Figure 6.4.

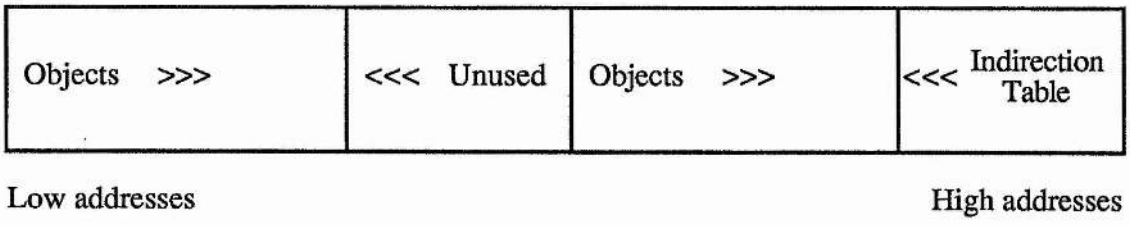


Figure 6.4. The heap of persistent objects.

Within the heap, all object addresses are represented by an index into the indirection table. These indices are the keys that are given to the main memory heap to represent an object's address. Indirect addressing was chosen to support an object copying garbage collector since it allows all references to an object to be changed by a single modification to the indirection table.

At any point in time, only one half of the heap contains objects and allocates space for new objects. When a garbage collection occurs all the reachable objects are copied to the other half of the heap. The two halves of the heap then reverse roles. However, the indirection table is always held at the high address end of the heap.

When an object is created, it is allocated space from one half of the heap and a single word from the indirection table. If necessary, the indirection table is extended. An attempt to create an object fails if the space it is allocated would interfere with the high address end of either half of heap. Once the space for an object has been allocated, the object's address is entered into the indirection table. The offset from the top of the indirection table to the entry for the object is used to represent the object's address.

The garbage collector always succeeds if it has sufficient space available to copy all the reachable objects. This can be achieved by keeping the unused part of the first half of the heap the same size as the indirection table. Thus, the maximum amount of space that can be allocated to objects in either half of the heap is the same.

6.3.2 The garbage collector

The garbage collection algorithm copies all the reachable objects from one half of the heap to the other half. For example, the garbage collector would transform the heap shown in Figure 6.5 into the heap shown in Figure 6.6. The garbage collection algorithm is composed of the following four steps:

a) reverse the role of the two halves of the heap,

b) copy the root object,

c) scan the half of the heap that will contain the reachable objects and

d) construct a free list in the indirection table.

6.3.2.1 Reverse the role of the two halves of the heap

The role of the two heap halves is reversed so that the normal object creation mechanism can be used while copying objects. At this point a flag is set in the heap's root object to indicate the first step of the garbage collection has been completed.

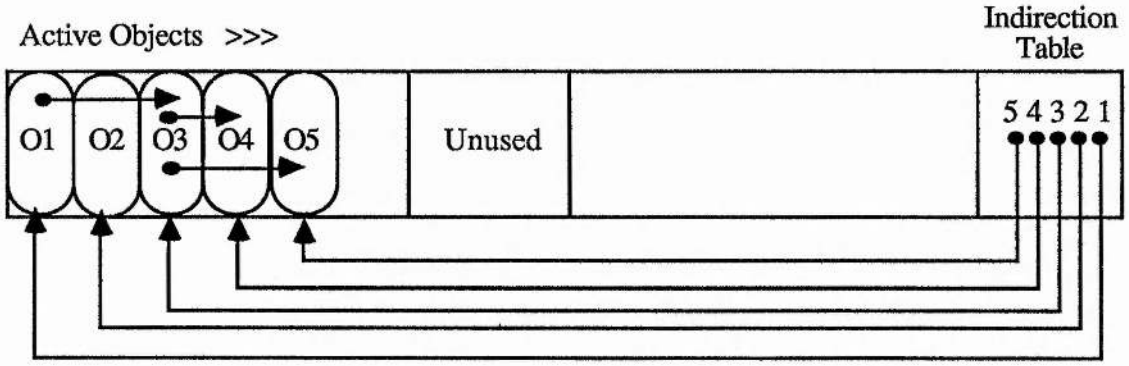


Figure 6.5. An example heap before garbage collection.

6.3.2.2 Copy the root object

The first object in the heap is copied to the other half of the heap and the object's indirection table entry is modified. To allow for the garbage collection being restarted after a checkpoint, the root object's stable store address is checked to see if it has already been copied.

6.3.2.3 Scan the half of the heap that will contain the reachable objects

The next step is to scan all the objects that are copied between the two halves of the heap looking for object addresses. While this is being performed, a flag is set in the root object to indicate the current phase of the garbage collection. In addition the address of the current object being scanned is also recorded. This information is sufficient to restart this phase of the garbage collection.

When an object is scanned, the indirection table entries for the objects it addresses are inspected. Any objects that have not yet been copied are copied and their indirection table

entries are modified. These copied objects will be subsequently scanned thereby copying the objects they address. At the end of the scan, every object reachable from the root object will have been copied.

6.3.2.4 Construct a free list in the indirection table

The final step of the garbage collection is to construct a free list in the indirection table. The purpose of the free list is to identify entries in the indirection table that no longer refer to reachable objects. These entries can be used to represent the addresses of new objects without the overhead of extending the indirection table. While the free list is being constructed, a flag is set in the heap's root object to indicate where the garbage collection should be restarted.

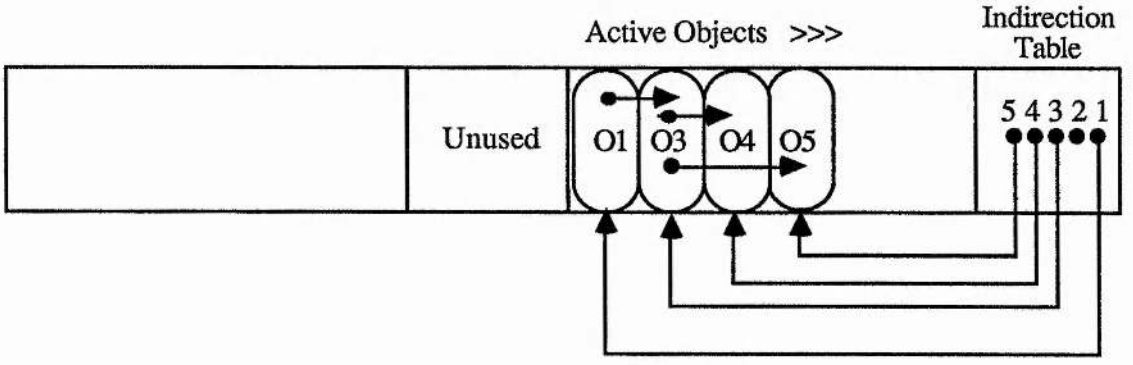


Figure 6.6. An example heap after garbage collection.

6.3.3 Checkpointing the heap of persistent objects

The heap of persistent objects is implemented directly on top of the stable storage layer. Therefore the heap's checkpoint mechanism can be implemented in two simple steps. Firstly, any data held outwith the heap must be replaced with the result that the entire heap will be held in stable storage. Secondly, the checkpoint procedure provided by the stable store's interface is invoked. This will checkpoint the stable storage and, therefore, checkpoint the heap of persistent objects.

6.3.4 Summary

Since the sole function of the layer just described is to implement a heap, its implementation is very simple. However, provided that an alternative implementation meets the specified interface there are no constraints on how sophisticated it may become.

6.4 The stable storage

The stable storage provides a contiguous range of virtual addresses that can always be restored to a self-consistent state. It also provides a checkpointing mechanism to change the self-consistent state to which the storage will be restored. The basic technique used to implement the stable storage is to make copies of those parts of the storage that are to be changed.

In the first implementation of the stable storage, the virtual address space is divided into pages with a page corresponding to a block of non-volatile storage. The stable storage operates by making copies of pages before they are changed. Thus, the state of the storage can be restored by replacing all the changed pages by their copies. To achieve a new self-consistent state, the appropriate changes are made and then all the copied pages are deleted. This prevents the stable storage being restored to a previous state. To ensure that the stable storage cannot be restored to an inconsistent state, the copied pages are deleted in a single atomic action. That is, regardless of when a failure may occur the copied pages will either all be deleted or none of them will.

6.4.1 Copying changed pages

A major influence on the implementation of the stable storage is the mechanism used to record copied pages. There are two basic techniques, maintain a mapping for every page or record only those pages that have been changed.

IBM's system R is an example of a system that maintains a mapping for every page[gra81,lor73]. The mapping is in the form of a page table. The entry for each page

contains the address of the original version of the page and the address of the current version of the page. When a page is first modified, a free page is allocated to hold the new version of the page. Thus, the original version of a page is never overwritten. A checkpoint of the system is performed by modifying the page table so that the current version of each page is viewed as the original version.

This particular implementation has the disadvantage of fragmenting large objects that cross page boundaries since modifications are not performed in place. Furthermore, it requires the stable storage to reserve sufficient space to hold the page tables. However, this solution has the advantage that its operation is independent of how many free pages are available to hold copies of changed pages. The number of free pages only affects the frequency with which a checkpoint was required.

Two alternative, but related, implementation techniques only record those pages that have been changed, they are known as a *before look* and an *after look* mechanism. A before look mechanism records the original copies of changed pages. When a failure occurs, the original copies of the pages are restored and the changes lost. In contrast, an after look mechanism records the new version of every page to be changed and only then modifies the original pages. When a failure occurs the original pages are overwritten by their new versions, thereby retaining the changes. These mechanisms operate by recording a list of changed pages together with a copy of those pages. An example of a before look mechanism that operates on objects instead of pages is used by the CPOMS.

A potential disadvantage of these mechanisms is the overhead required to identify pages that have already been changed and copied. In the case of systems such as the CPOMS, this is of little importance since only a relatively small number of changes are made between checkpoints. Hence, a small mapping table is sufficient to support the mechanisms. However, if the entire stable storage was to be changed between checkpoints then the mapping table would be equivalent in size to a page table. Thus, the before look and after

look mechanisms are better suited to stable stores that undergo relatively few changes between checkpoints.

The copying technique used in the first implementation of the stable storage is a cross between the two techniques just described. It is similar to the first technique in that it maintains a mapping for every page and it is similar to the second technique in that the mapping only indicates whether or not a page has been copied.

6.4.2 The layout of the stable storage

The stable storage is divided into two halves, the first half containing the current version of a page and the second half containing copies of changed pages. This division of the stable storage has several advantages. Firstly, the copy of a page in the first half of the store is held at the same relative position in the second half of the store. Thus, it is not necessary to explicitly record the location of a page's copy. A second advantage is that a single bit is sufficient to record whether or not a page has been copied. The division also allows the current version of every page in the stable storage to be changed between checkpoints.

A disadvantage of the division of the stable storage is that only half the store is available to hold useful data. Furthermore, if checkpoints are relatively frequent then only a few of the pages in the second half of the store will be in use at any one time. Hence, the division of the stable storage may waste a significant fraction of the available storage. This disadvantage was disregarded in the first implementation of the stable storage since the resulting mechanism is both simple and efficient.

The first half of the stable storage is divided into three sections, a bitmap that records any copied pages, a pair of root pages[atk83b] and that part of the stable storage that is made available to the heap of persistent objects, as shown Figure 6.7.

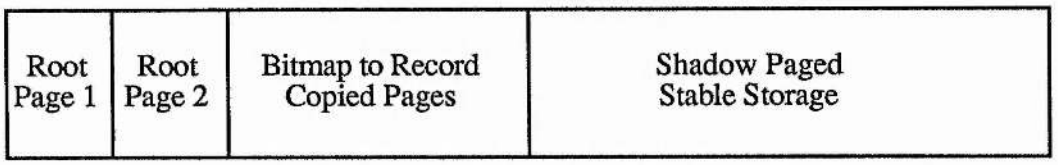


Figure 6.7. The first half of the non-volatile store.

When a page in the first half of the stable store is to be modified, a copy of the page is written to the second half of the store. The relative position of the copied page in the second half of the store is the same as the relative position of the original page in the first half of the store. The appropriate bit in the bitmap is then set to record the fact that a copy has been made. When the page is subsequently modified, a check of the bitmap will prevent further copies being made.

In addition to preventing multiple copies, the bitmap is also used to restore the stable store to a self-consistent state. This can be achieved by scanning the bitmap and replacing any modified pages by their copies. However, this is only possible if the bitmap is up to date.

To ensure that the bitmap is correct, it must be updated and written to non-volatile storage before a modified page is written but after that page has been copied. Thus, if the bitmap indicates that a page has been copied there will be a corresponding copy in the second half of the stable store. This requires a checkpoint mechanism to be applied to the bitmap to prevent it being corrupted by a failure while it is being written.

6.4.3 The root pages

The additional checkpoint mechanism is provided by a root page, two copies of which are held at the start of the stable storage. When the current value of the page is written to non-volatile storage, the older of the two copies is overwritten. To allow the copies to be differentiated, the root page has a date stamp value at either end, as shown in Figure 6.8. The copied root page with the higher time stamp is the most recent.

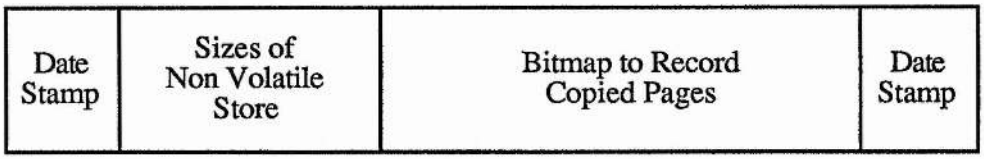


Figure 6.8. The layout of a root page.

The root page records the dimensions of the non-volatile storage such as its length in blocks, its half length and the number of pages that are allocated to the bitmap. The root page also contains a root bitmap that has a bit for every page holding the main bitmap. The root bitmap is used in the same way as the main bitmap. That is, before a page in the main bitmap is written it is first copied. Once the copy has been successfully written the page's bit in the root bitmap is set. The root page is then written followed by the modified page in the main bitmap. The final use made of the root bitmap is to checkpoint the main bitmap. That is, the modified state of the main bitmap must be preserved in case it is required to restore the stable store. This is achieved by clearing the root bitmap and then writing the root page to non-volatile storage.

As an optimisation, the main bitmap can be discarded if the root bitmap is large enough to contain a bit for every page in the first half of the stable store. When this optimisation is used, the checkpoint mechanism applied to the main bitmap can be adapted for use by the entire stable store. The only significant change to the mechanism is that the root bitmap is not cleared until the entire stable store has been checkpointed. Thus, for small stable stores only one level of checkpoint mechanism is required.

The root pages' date stamps are used to detect corruption. The interface to the non-volatile store guarantees that a block is written in a predefined order and that a write only completes if successful. That is, if a root page is successfully written to non-volatile storage then the date stamp values at its start and finish are the same.

When a write causes corruption, the write fails and the non-volatile store causes the architecture to halt, as described in Section 6.5. Hence, the stable store can normally operate under the assumption that no corruption will occur. However, it must check for corruption when the stable store is started since the store may have stopped as the result of a failure. Any corruption can be detected by inspecting the date stamps in the two copies of the root page. The copy with the highest date stamp contains the most recent version of the root page. However, if that page's date stamps are not the same then it was corrupted and the other copy is used. Provided that the non-volatile store is not subject to a catastrophic failure, the other copy is free from corruption.

It is only necessary to check the root pages for corruption for the following reasons. A failure in the normal operation of the stable storage may corrupt at most a single page. Consequently, the corruption of a page's copy can be ignored since the bitmap entry for the page would not have been recorded when the failure stopped the stable store. Similarly, the corruption of a modified version of a page can be ignored since the main bitmap will contain a reference to the page's copy. Hence, the restoration of the stable store would automatically overwrite the corrupt page with its copy. Finally, the corruption of the bitmap itself can be ignored since it will be restored by the checkpoint mechanism supported by the root pages.

6.4.4 Checkpointing the stable store

The self-consistent state recorded by the stable store can be altered using the checkpoint procedure provided by its interface. A checkpoint is performed in three steps:

a) writing every modified page to non-volatile store,
b) eliminating the previous self-consistent state, and
c) clearing the root bitmap.

6.4.4.1 Writing every modified page to non-volatile store

Firstly, every modified page must be written to the non-volatile store. This step involves the copying of modified pages and the updating of the two bitmaps as described above. Once this step has been completed the non-volatile store will contain all the modified pages and a copy of each modified page. That is, the non-volatile store contains a new self-consistent state together with sufficient information to restore the stable store to the previous self-consistent state.

6.4.4.2 Eliminating the previous self-consistent state

The next step in the checkpoint is to eliminate the previous self-consistent state. This is achieved by clearing all the entries in the main bitmap, thereby discarding all the copies of modified pages. To prevent the stable store being left in an inconsistent state, this step must be performed as an atomic action. Thus, when a page of the main bitmap is modified the page is copied, the root bitmap is updated, the root page is written to non-volatile storage and only then is the modified page written to non-volatile storage. However, unlike the checkpoint mechanism described above, the root bitmap is not cleared.

6.4.4.3 Clearing the root bitmap

At this point in the checkpoint a failure would cause the main bitmap to be restored to its state at the end of the first step. As a consequence, all the previously modified pages would also be restored causing the stable store to revert to its previous stable state. Therefore, to complete the checkpoint the root bitmap is cleared and then written to the non-volatile store.

6.4.5 Summary

The shadow paging mechanism just described is an example of a simple implementation of the stable storage architecture layer. It provides the ability to incrementally modify the entire stable store between checkpoints and it can be easily optimised. For example, when a modified page must be written back to non-volatile storage, all the modified pages could be written back. Also, if the modified pages have already been copied, the act of updating the

main bitmap can be delayed until the page is written out. Hence, the writing of the pages and the updating of the main bitmap could be performed as a single operation rather than being repeated for each individual page.

6.5 Non-volatile storage

The non-volatile storage layer must provide the stable storage layer with a fixed set of contiguous equal sized blocks. Furthermore, the operations on these blocks must conform to a specified set of assumptions. These are, a block can only be corrupted if it is being written to, if a write operation completes it did not cause any corruption, all blocks are written in a specific address order, reading a block never causes corruption and if a read or write fails the architecture halts. These assumptions are necessary to support the stable storage mechanism just described.

In the first implementation of the architecture the non-volatile storage is provided by a single fixed size UNIX file[tho78]. This implementation strategy is based on the assumption that the UNIX file system is block oriented and conforms to the required assumptions. In practise, these assumptions may not be valid but this limitation may be overcome by duplicating the file over several physical devices. In fact, it is possible to provide an arbitrarily stable store by maintaining multiple physically disjoint copies of the file[koh81]. Furthermore, each of the necessary assumptions could be implemented by procedures that would coordinate access to the multiple copies. However, for the purposes of experimentation, a single file supported by regular dumping is sufficient.

6.6 Conclusion

The implementation of the layered architecture described above has been constructed on SUN workstations[sun86a] running the SUN version of the UNIX operating system[sun85]. The UNIX system is used to provide the necessary primitives to access the available disk storage, to perform graphics operations, to perform floating point operations and to permit access to network software. This use of a host operating system is purely for

convenience. In fact, it only significantly affects the implementation of the non-volatile storage layer and some components of the abstract machine. Furthermore, the specified interfaces for each of the architecture layers are independent of how any particular primitives are implemented.

The extension of the layered architecture to include an optional local heap allows the architecture to be hardware independent. For example, the implementation of the local heap by an adaption of the CPOMS technology provides a relatively efficient software implementation of the architecture. Thus, the architecture can be effectively implemented without the need for costly special purpose hardware. This permits the architecture to be ported to a wide range of existing computer systems. Since the choice of host operating system only significantly affects the abstract machine and the non-volatile storage, the cost of porting the architecture should be relatively low.

The overall complexity of the layered architecture's implementation is considerable. However, the modularity of the layered architecture allows abstraction over the complexities of the implementation techniques. These abstractions support experiments with the implementation of individual architecture layers. Provided that a layer meets its specified interface, it may be reimplemented without affecting the other layers.

The independence of the different layers coupled with the potential for an efficient software implementation make the layered architecture a powerful and flexible tool for experimentation.

7 Support for distribution

One design aim of the layered architecture is to support experiments with distribution. The provision of distribution requires solutions to a wide range of problems many of which arise from the combination of disparate computer architectures[koh81,tan85]. These problems include:

a) What form of addressing mechanism should be supported?
b) How should different physical architectures communicate data?
c) Should objects be moved between computers?
d) What consistency constraints should be applied to a network?
e) Can distributed garbage collection be supported?
f) How are failures of network components to be handled?

This chapter describes how the layered architecture supports distribution and presents an example of a distribution mechanism that may be constructed from that support.

7.1 Support for addressing mechanisms

There are a wide variety of addressing mechanisms that can be employed by a distributed system. For example, a global addressing mechanism may be adopted that gives every object a unique, network wide, address. Alternatively, an object could be represented by a two part address that identifies a particular computer and an address within that computer. More complex addressing schemes exist that limit the interpretation of an address to a local context. With such an addressing mechanism, an object is identified by a chain of context dependent addresses that corresponds to a route through the network to the addressed object.

To permit flexibility in the choice of addressing mechanism, the layered architecture does not provide any explicit support for remote addressing. Instead remote addresses may be represented by persistent objects whose interpretation is the responsibility of an individual

implementor. This permits arbitrarily complex addresses to be constructed without affecting the implementation of the internal addressing mechanisms of an individual architecture.

7.2 Support for communicating data

For the purposes of experimentation, the correct interpretation of data may be achieved by defining a single architecture that is emulated by each physical architecture. This involves defining a range of data formats, defining the interpretation of the data formats and defining an instruction set for executable code. The strict emulation of a common architecture may be extremely expensive even for apparently similar physical architectures. To reduce the effect of this problem the layered architecture has adopted the following solution:

Firstly, all persistent objects are defined in terms of integer words and are communicated using an agreed integer representation. Secondly, the data formats supported by the architecture are defined by the mapping of PAIL onto persistent objects and, finally, a common abstract machine is defined whose executable code is also mapped onto persistent objects. Thus, similar physical architectures that use the same size of integer can represent exactly the same data, with exactly the same interpretation.

This solution to the problem of communicating and interpreting data should not result in an undue loss of efficiency for operations on the majority of data formats. However, the implementation of certain data formats may vary considerably between physical architectures. To accommodate this, the layered architecture makes special provision for the efficient interpretation of the following data formats.

a) character strings,

b) floating point numbers,

c) raster graphics images and

d) executable code.

7.2.1 Byte ordering

Each persistent object manipulated by the abstract machine can be defined in terms of integer words. However, this may not yield the most efficient representation of certain objects. For example, the abstract machine's instruction set is a byte stream. It may be more efficient for an interpreter to process the bytes in address order rather than have to reorder individual integers to yield the appropriate bytes. Similarly, an object representing a string of characters may be more efficiently operated on if the characters are held in address order.

The provision of objects requiring a specific byte ordering presents a problem if such an object is moved between computers with different integer representations. When this occurs every integer word will have its bytes reordered. Hence, the object may not be interpreted correctly on some computers. To overcome this problem it is necessary to record the byte ordering of the computer that created the object.

In the first implementation of the layered architecture, this is achieved by using two marker bits to identify the byte ordering within an object. The marker bits are held in the first word of an object, as described in Section 6.2.2. One bit is used to identify the ordering of the two 16 bit components of a 32 bit integer and the other bit is used to identify the ordering of the two 8 bit bytes within a 16 bit component. When an object is created it is tagged with the byte ordering of the computer that created it.

To complement the tagging of objects, the abstract machine also checks the byte ordering of particular objects. The checking is necessary since the byte ordered parts of an object are only known to those abstract machine instructions that manipulate the object. For this reason it may not be possible to automatically change the ordering while an object is being moved between computers. Since there are only four variations in the representation of an integer, the tagging is sufficient to identify the transformation required to reorder the bytes within an object.

A major advantage of this solution is that it does not affect the storage and manipulation of a persistent object. It only affects how particular objects are interpreted.

7.2.2 Floating point numbers

Floating point numbers can be represented and manipulated by a wide range of different mechanisms. For example, a VAX supports several different formats of floating point numbers[pay80]. Each format allows a floating point number to be of several different lengths. Consequently, the result of a floating point calculation may vary depending on the length of the representation used. In a heterogeneous environment the number of possible variations may increase dramatically as different physical architectures are added.

To eliminate the variation in floating point representations the first implementation of the layered architecture has adopted the IEEE 754 standard for binary floating point numbers[coo81,cod81,hou81,iee81]. The chosen representation is further restricted to be 64 bits in length with all arithmetic being performed in the 80 bit extended format.

The IEEE standard was chosen for several reasons. Firstly, the representation of an IEEE floating point number can be defined in terms of bit significance within two integers. This allows floating point numbers to be moved between computers as pairs of integer values without the need for special attention.

A second reason is that implementations of the IEEE standard are available as software libraries[app86b], floating point coprocessors[mot85] and floating point accelerators[sun86b,sun86c]. In fact, the first implementation of the architecture has been constructed on a Motorola MC68020 based system with a floating point coprocessor[sun86a]. The availability of hardware implementations of the IEEE standard should minimise any performance overheads involved in adopting a single format for all floating point numbers.

7.2.3 Raster graphics

The variety of representations and implementations of raster images is similar to that for floating point numbers. However, the majority of differences in implementation are small enough to allow a common format to be operated on by several different implementations. As an example of this, the PS-algol graphics implementations on the ICL Perq[icl83], the Apple Macintosh[app86a] and the SUN use an almost identical format[bro86]. In all three cases the raster operations are performed by the host computer's raster graphics procedures but only the Perq implementation has access to hardware and firmware support. The SUN and Apple Macintosh make use of software libraries.

Since hardware support is extremely specialised in nature, the abstract machine provides access to a software library that implements a complete set of raster graphics procedures. The library procedures include the 16 raster combination procedures and a line drawing procedure. The format operated on by the procedures is defined entirely in terms of bit significance within integers. Thus, the provision of the software library allows raster images to be moved between computers without altering their interpretation.

7.2.4 Executable code

Executable code is one part of a physical architecture that is guaranteed to be unique to a particular computer. It is usually directly supported by special purpose hardware and is finely tuned to its host architecture. For example, in decoding an instruction, a computer may decompose the instruction into several distinct components in a single operation. A similar decomposition of the same instruction on a different computer may have to be done serially. For this reason, it is not feasible to force all physical architectures to emulate a particular physical architecture's instruction set. However, some form of compromise solution is required if executable code is to be moved between physical architectures and then be executed.

There are two alternative solutions provided within the layered architecture. One alternative is to use the PAIL representation of a procedure and the other is to use a predefined abstract machine code.

The PAIL solution requires a procedure to be recompiled whenever it is to be executed on a different physical architecture. An advantage of this solution is that the resulting executable code will be tailored to suit the features of the architecture on which it will be run. However, there are several disadvantages to this scheme. Firstly, the recompilation process may be time consuming. Secondly, the procedure's PAIL representation may be quite substantial particularly if the procedure's environment is included. Finally, all the PAIL associated with the procedure will need to be moved to the computer on which the compilation process will take place. Thus, the PAIL solution can be expensive in time, space and network traffic.

The solution adopted by the first implementation of the layered architecture is to use a single predefined abstract machine code. The abstract machine code chosen was designed to directly support the efficient execution of PAIL programs. As such, many of the abstract machine instructions correspond to short sequences of the instructions provided by a physical architecture. This has the two-fold effect of reducing the overheads involved in interpreting the abstract machine code as well as providing a compact representation of the corresponding PAIL.

The chosen abstract machine code is defined in terms of a byte stream[bro88a]. Each instruction is represented by a single byte operation code followed by an appropriate number of bytes to represent any parameters. The interpretation of an instruction's parameters is strictly defined in terms of byte order. Since the abstract machine code is both compact and does not require parallel decoding it can be efficiently interpreted on any physical architecture. The only constraint that must be enforced is the preservation of the byte ordering.

When an object representing abstract machine code is moved from one computer to another the byte ordering may be altered. To allow for this reordering, all executable code is checked before it is executed. The check involves comparing the byte order with which the code's object is tagged and the byte order for the computer on which the code is to be executed. If the two byte orders differ the executable code is reordered within its object and the object's tag bits updated. As a direct consequence of this, the code object will appear to have been created on the computer on which it is executed.

7.2.5 Summary

The communication support just described is suitable for a wide range of physical architectures that use the same size of integer. It provides a common architecture to which every instance of the layered architecture conforms. Consequently, a distribution mechanism constructed from the support is free from the problems of correctly interpreting data and need only concern itself with problems such as, what form of addressing mechanism should be supported, should objects be moved between computers, what consistency constraints should be applied to a network, can distributed garbage collection be supported and how are failures of network components to be handled?

7.3 A distribution mechanism for a tightly coupled network

In order to demonstrate the effectiveness of the support for distribution, a possible distribution mechanism will now be described. It should be noted that this description does not relate to an actual implementation. However, the support described in Section 7.2 has been implemented with the exception of the checks required to establish byte ordering.

The proposed distribution mechanism implements a very large persistent store by allowing several instances of the layered architecture to be viewed as a single instance. As such, the implementation the distribution mechanism must provide:

a) a remote addressing mechanism,

b) a distributed checkpointing mechanism and

c) a distributed garbage collection mechanism.

To simplify the example, the distribution mechanism does not include any facilities for the distributed execution of programs. It is envisaged that such facilities are provided by the concurrency mechanisms that are used by the architecture instances. Similarly, there are no facilities to control the movement of objects between computers. The details of how this may be achieved are the subject of further research and outwith the scope of this thesis.

To further simplify the example, it is assumed that the distribution mechanism will only be used on a tightly coupled network. A tightly coupled network is one where all the non-volatile storage is held on a server and can be treated as an atomic unit. Thus, the dumping and recovery strategy for the network can be the same as for a single computer. Also, the non-volatile storage in the network can be maintained in a self-consistent state. This has two important consequences for the design of a distribution mechanism. The treatment of failures can be simplified and a distributed garbage collector can be implemented.

The treatment of failures can be simplified in the following ways. A failure of an individual node in the network can be treated as a total system failure that causes the entire network to be restarted. This would free the distribution mechanism from the need to deal with problems such as the partitioning of the network due to communication failures. A further simplification can be achieved by restoring the entire non-volatile storage if any one of its components becomes corrupted. This is possible because all the network's non-volatile storage can be managed as a single atomic unit.

The implementation of a distributed garbage collector requires all the components of the non-volatile storage to remain in a self-consistent state. That is, no individual component of

the store, that is garbage collected, can revert to a previous state independently. This ensures that a component of the store does not restore references to objects that were deleted by a garbage collection, unless the referenced objects are themselves restored. In an arbitrary network there may be specific constraints such as the scale of the system that force components to be operated independently. Hence, a distributed garbage collector is only feasible in a specific class of distributed system.

7.3.1 The distribution layer

The model of distribution supported by the example mechanism is intended to be transparent to all PAIL programs. That is, there will be no facilities in PAIL that allow a program to determine the location of a persistent object. Hence, the distribution layer should not be visible to the PAIL layer of the architecture.

The abstract machine layer of the architecture is ultimately responsible for implementing all operations that are permitted on a persistent object. Consequently, since PAIL is unaware of the distribution layer, the abstract machine must be responsible for implementing the distribution. The lower layers of the architecture cannot provide the distribution because their sole purpose is to store persistent objects and not to operate on them.

There is one problem that does require a PAIL program to know the location of a persistent object. In order to access objects from another instance of the architecture, it is necessary to discover the address of the root object of the other instance. Thereafter, the desired objects can be accessed by following the address fields of objects reachable from the root object. This problem only arises when a new instance of the architecture is added to a network. To overcome this problem the abstract machine provides PAIL with a lookup procedure to find the root object of another architecture instance. The procedure need only be used when a new instance is added to the network, thereafter the distribution mechanism can reman hidden.

7.3.2 Addressing remote objects

The most important part of the distribution mechanism is the support provided for addressing remote objects. As described in previous chapters, the address fields within an object cannot reference a remote object. To overcome this problem the following indirection mechanism has been adopted.

The addresses of objects held on other instances of the architecture are represented by special objects known as *link objects*. A link object contains three pieces of information, the address of a type description for the link object, the name of the architecture instance that owns the object and a key value, see Figure 7.1. The key value is local to the instance of the architecture that owns the object.

A type description is recorded within the link object because by convention all objects manipulated by the abstract machine contain a type description. Although this type description will not be used, it is included for completeness.

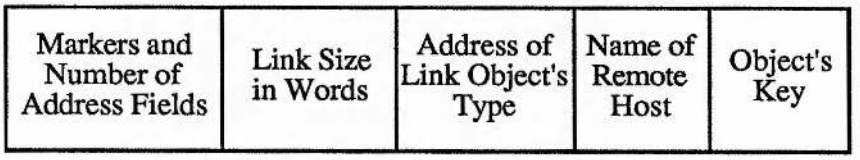


Figure 7.1. A link object that represents a remote object.

An important issue in any distribution mechanism is the routing of messages between the nodes of a network. For the purposes of this example, it is assumed that the network software provided by the layered architecture's host environment will perform this function. Hence, it is possible to simply record the name of a remote node and delegate the routing of messages to the host network. As a direct consequence of this routing strategy, all remote addresses can be thought of as direct references to the remote object.

The key held within a link object uniquely identifies the addressed object within its host architecture instance. However, the key should not be an object address since object addresses may be modified by a garbage collection, requiring all copies of an object's address to be modified, which could prove extremely costly .

Link objects are created when object addresses are passed between network hosts. An address is passed as a pair consisting of the object's key and the name of its host architecture instance. To support the exchange of object addresses, the distribution mechanism maintains two lists, an export list and an import list.

The export list is used to record addresses that have been passed to other network hosts. It is organised so that it can be searched using an exported key value or an object address. The import list is used to record object addresses that have been received from remote hosts. It is organised to allow it to be searched using a combined host name and key value.

Figure 7.2 shows how the two lists are used to record the address of an object that was passed between two hosts, host A to host B. The import list entry on host B is the link object that represents the object's remote address whereas the export list entry on host A contains the object's actual address.

The maintenance of the import and export lists must take account of the following four cases that may arise when object addresses are passed across the network,

a) exporting the address of a local object,
b) exporting the address of a remote object,
c) importing the address of a local object and
d) importing the address of a remote object.

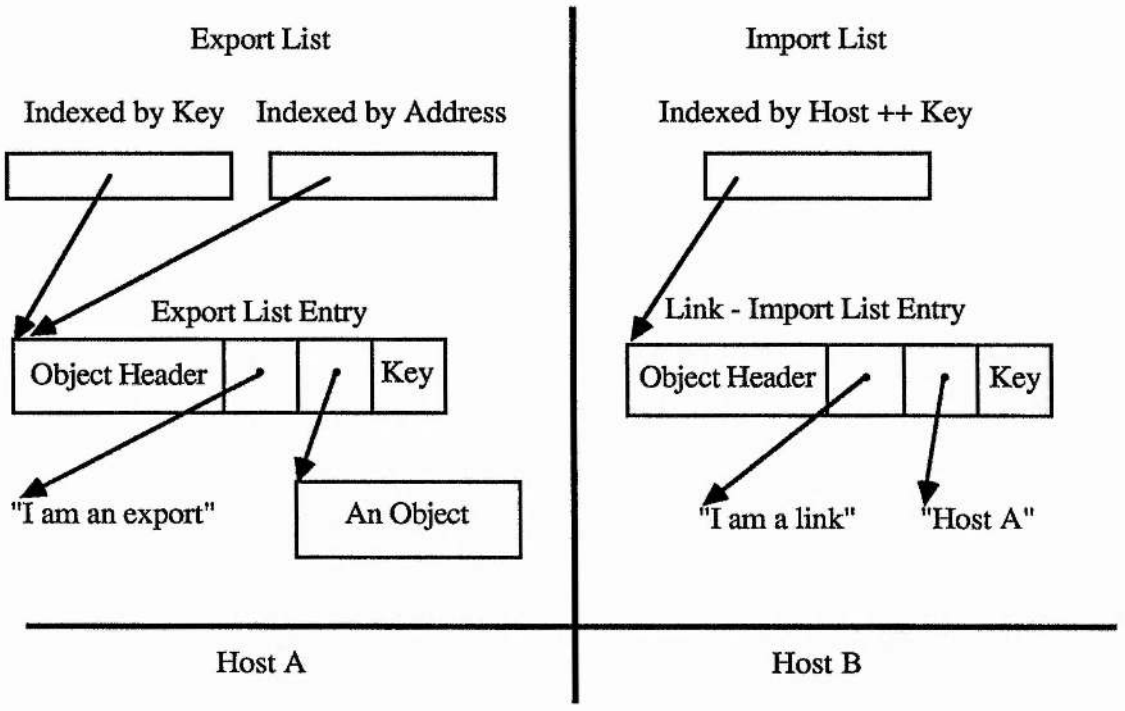


Figure 7.2. The import and export lists used to address an object on host A from host B.

7.3.2.1 Exporting the address of a local object

When the address of a local object is exported, a key value must be assigned to the object. In order to avoid allocating more than one key per exported address the export list maintains a record of all key values that have been issued. As a result, a new key value need only be allocated the first time an object's address is exported. Thereafter, the key can be found by searching the export list.

To support the elimination of multiple keys it is necessary to index the export list by object address. Since object addresses may change whenever a garbage collection occurs, the organisation of the export list may need to be periodically re-evaluated.

In addition to the elimination of multiple keys, the export list is also used to preserve exported objects over local garbage collections. All exported objects are reachable from the export list and are therefore reachable from the local root object. At some future point an

exported object may only be reachable from its local root object via the export list but it may still be reachable from a remote root object. The export list causes such an object to be retained until a network wide garbage collection determines that it is no longer reachable.

7.3.2.2 Exporting the address of a remote object

The transmission of the address of a remote object can be performed by simply sending the host name and key value held in the address's link object. No further action is required from the sender.

7.3.2.3 Importing the address of a local object

All addresses are received from the network in the form of a host name and key value. When an address is received with the same name as that of the receiving host, the key value refers to the address of a local object. Hence, the action required is to map the key value to the appropriate object address. This can be performed by searching the export list using the key value. Since all object addresses are recorded before an address is exported, the search of the export list will always succeed. The search requires the export list to be indexed by key value. In contrast to object addresses, key values are fixed. Hence, the key value index of the export list never requires re-evaluation after events such as garbage collection.

7.3.2.4 Importing the address of a remote object

When the address of a remote object is received from the network, a link object must be created to represent it. Remote addresses are unique and it is desirable to only create one link object per remote address. Thus, the first step in receiving a remote address is to search the import list. The import list records all remote addresses that have been received. To support searches for remote addresses, the import list may be indexed by both host name and key value. The first time a remote address is received, the import list will not contain a corresponding link object. In this case a new link object is created and the host name and key value are entered into it. In turn, this link object is entered into the import list for future use. Once a link object for the remote address has been created or found, the address of the

link object is used to represent the remote object. It should be noted that this directly supports address identity since all references to a remote object on a particular architecture instance will be represented by the same local object.

7.3.3 Operations on remote objects

Whenever the abstract machine dereferences a persistent object, it must be able to determine whether or not the object is a link object. To support this, all link objects are marked using one of the eight marker bits in their first word to indicate their special purpose, as described in Section 6.2.2. When a link object is encountered there are two possibilities; either the operation is performed on the local host or on the remote host.

The majority of operations performed by the abstract machine can be decomposed into reading or writing integers and object addresses. As such, these operations can be performed on the local host with the appropriate read and write operations being forwarded to the remote host. Similarly, operations that require a specific byte ordering within an object can also be performed locally. However, these operations must ensure that integer values read from or written to remote objects are reordered correctly.

The appropriate reordering can be achieved by using two pieces of information. Since, all operations performed by the abstract machine are generated from PAIL, the abstract machine knows the kind of object it should be manipulating. This will determine which sections of an object require to be in a specific byte order. To complement this, the first word of the remote object is tagged with the byte ordering within its integers. Using this information, the appropriate reordering of the bytes within an integer can be performed after an integer is read and before an integer is written.

There are a specific set of abstract machine operations that perform tasks peculiar to the local architecture instance. For example, to read information from a file requires the file being read to be accessible to the abstract machine. This would not be the case if the I/O operation

was performed by a different architecture instance. Hence, operations on remote objects such as file descriptors must be performed by the abstract machine that created the file descriptor.

7.3.4 Distributed checkpointing

The distributed checkpointing mechanism is provided by the implementation of a two phase commit protocol[lam76]. The protocol ensures that either every component system within a network performs a checkpoint at the same time or none of the systems perform a checkpoint. The protocol would operate as follows:

1. Record the start of a distributed checkpoint.
2. Perform a checkpoint operation to a point at which it may either be abandoned or completed.
3. Instruct all other systems within the network to perform the previous step.
4. Check that the other systems have performed the requested operation. A failure in any of these first four steps will result in every system in the network being instructed to abandon the checkpoint.
5. Record the successful completion of the first part of the checkpoint.
6. Complete the checkpoint operation. Any failure subsequent to this step of the algorithm will cause the algorithm to restart from step 7.
7. Instruct all other systems within the network to complete their checkpoints.
8. Check that the other systems have completed their own checkpoints.
9. Record the successful completion of the checkpoint.

To support the two phase commit protocol, the interface to the stable storage is extended by the two procedures, *prepare_checkpoint* and *abort_checkpoint*. The operation of prepare_checkpoint is identical to the existing *checkpoint* procedure except that the previous self-consistent state is not destroyed. As a result the stable storage may contain two self-consistent states, one saved by prepare_checkpoint and one saved by checkpoint. This

allows the decision to complete or abandon a checkpoint to be delayed without the risk of losing data due to a failure. The prepare_checkpoint procedure is used to implement step 2 of the above protocol.

Once a checkpoint has been prepared, it may be completed using the existing checkpoint procedure or it may be abandoned using the abort_checkpoint procedure. When a checkpoint is abandoned the stable storage is restored to the self-consistent state saved by the last use of the checkpoint procedure.

Since the distribution mechanism is provided at the level of the abstract machine, the additional checkpoint procedures are also made available as part of the interface to the persistent heap.

7.3.5 Distributed garbage collection

Within the example distributed system, garbage collection is provided at two levels, locally within individual systems and globally over the entire network. Since the import and export lists are composed of reachable objects, a local garbage collection can never discard an object whose address has been exported. However, it can be used to identify a significant proportion of a system's unreachable objects thereby reducing the frequency with which a full scale distributed garbage collection is required.

To complement the use of local garbage collections, the example system also supports a garbage collection algorithm that can identify every unreachable object in a network. A key feature of the garbage collection algorithm is that it does not take account of the import and export lists in determining the reachability of an object. As a result, this algorithm will eliminate all unreachable objects within a network, including circular lists of objects that traverse more than one component system. The algorithm operates as follows:

1. Each system in the network marks every object reachable from their own root objects. When a link object is marked, the marking is propagated to the addressed object's system. The import and export lists are not used for marking so that exported objects are only marked if they are reachable via another route.
2. The import and export lists are scanned and all unmarked entries are removed.
3. The import and export lists are marked since they are reachable from their own system's root object. When this step is complete every object reachable from a root object will have been marked.
4. Each system in the network deletes any unmarked objects it may contain.

To implement the algorithm just described the garbage collector must be able to recognise link objects, interpret link objects, mark objects, identify marked objects and delete unmarked objects. The first two operations are provided as part of distribution mechanism whereas the remaining operations are provided as part of the persistent heap.

To allow the distribution mechanism to remain distinct from the persistent heap, the persistent heap provides the following additional interface procedures:

a) Mark_object,

This procedure is used to mark an object and every other object reachable from it.

b) Marked?,

This procedure is used to test if a specified object has been marked.

c) Collect.

This procedure is used to collect and destroy any unreachable objects.

Given these additional interface procedures, the abstract machine can implement the garbage collection algorithm described above. However, the persistent heap is unable to propagate the marking to a remote system if it encounters a link object since it cannot recognise or

interpret a link object. To overcome this problem, the distribution mechanism provides the persistent heap with the following two procedures:

a) Link?,

   This procedure is used to test if a specified object is a link object.

b) Mark_link.

   This procedure is used to mark the object addressed by a link object. It will cause the mark procedure in the remote persistent heap to be invoked.

Given these interface procedures the distributed garbage collection can be implemented while maintaining the independence of the distribution mechanism and the persistent heap. That is, provided they preserve their specified interfaces, the distribution mechanism and the persistent heap may be reimplemented independently of each other.

7.3.6 Summary

The example distribution mechanism described above, supports the addressing of remote objects, operations on remote objects, distributed checkpointing and distributed garbage collection. It has been designed to be used as part of the abstract machine. As such, PAIL programs are not aware of the locality of their data. This ensures that the resulting distributed system preserves the persistence abstraction. The overall structure of a system constructed using the example mechanism is shown in Figure 7.3.

7.4 Conclusion

The layered architecture has been designed to support experiments with distribution over networks of similar physical architectures. The support includes a flexible architecture definition that can be efficiently mapped to different physical architectures that use the same size of integer. As part of this definition, the formats and interpretation of floating point numbers, raster images and executable code have been specified with a view to their efficient implementation. The overall aim of the distribution support is to allow the problems

of combining different physical architectures to be separated from those of modelling the distribution.

To demonstrate the feasibility of the support, a simple distribution mechanism has been described. It implements a distributed persistent store that behaves as if it were a single instance of the layered architecture. As such it provides remote addressing, distributed checkpointing and distributed garbage collection.

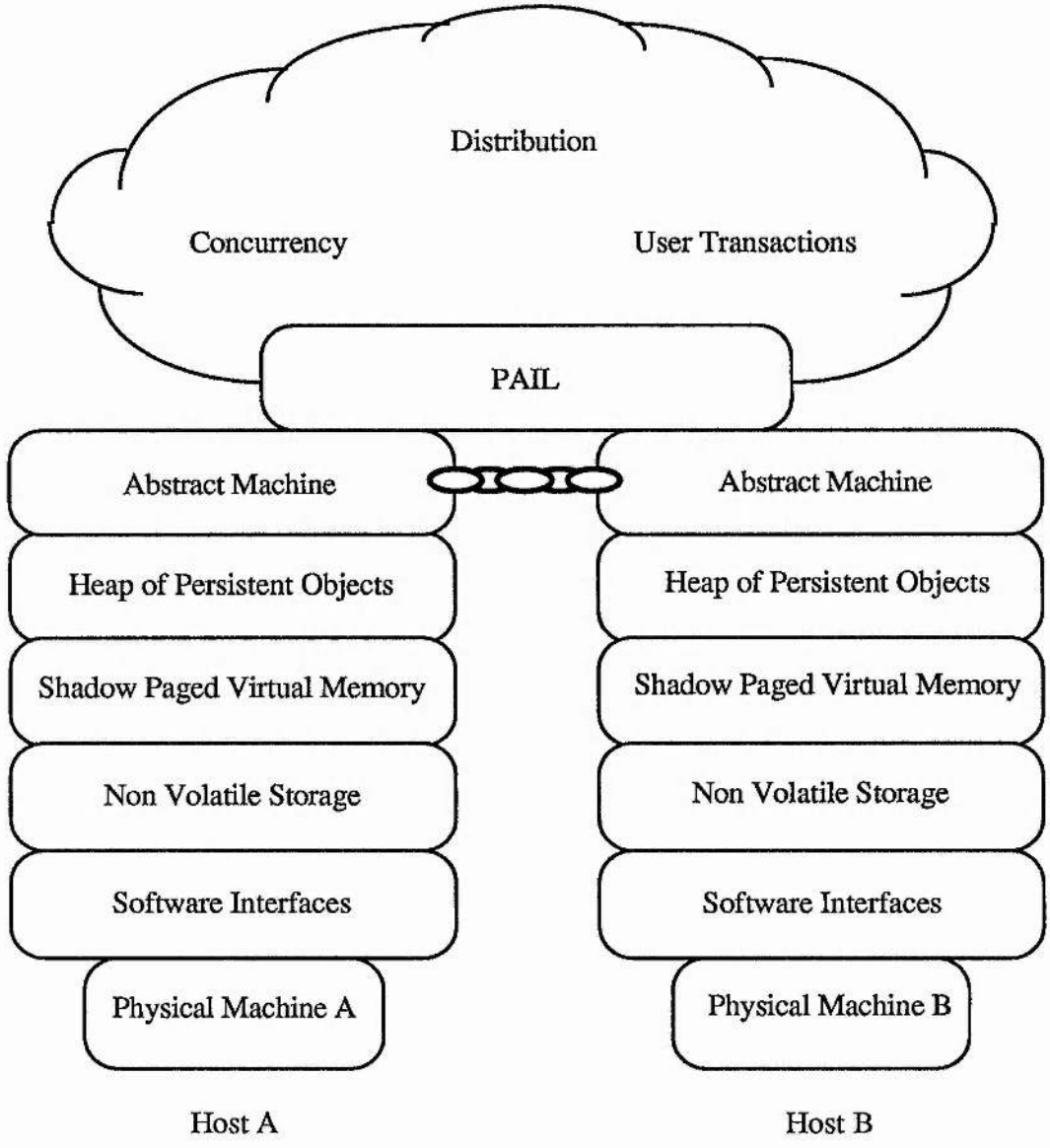


Figure 7.3. The architecture layers and their connections between two hosts.

8 Dynamic storage configuration

This chapter describes how the behaviour of the architecture may be dynamically configured to reflect the implementation of the stable store. The stable storage layer provides a contiguous area of storage that can always be restored to a self-consistent state. To operate correctly, the stable store must provide sufficient additional storage to record any changes that may be made between checkpoints. The additional storage is referred to as shadow store since it is assumed to contain shadow copies of the changed areas of stable store. If the shadow store is exhausted it may not be possible to restore the previously recorded state.

To permit flexibility in the choice of implementation strategy, the interface to the stable store does not specify how much of the stable store may be changed between checkpoints. This allows an implementor of the stable store to determine how much shadow store is required with regard to the intended use of the implementation. For example, one system may be provided with sufficient disk storage to support a large shadow store. Another system, with only a small amount of disk storage, may need to use a higher proportion of its available storage for active data rather than for the shadow store.

Since the amount of shadow store provided by an implementation of the stable store may be limited, the interface to the stable store includes two procedures, *page size* and *shadow size*, that allow the architecture to monitor its use of shadow store. The architecture is then able to anticipate the exhaustion of the shadow store and, if appropriate, to invoke a checkpoint.

All modifications to the stable store are, in fact, modifications to the non-volatile store. The non-volatile store is accessed as a sequence of equal sized blocks that are read and written as entire blocks, as described in Section 4.3.3. Thus, a modification to part of a block causes the entire block to be re-written. To reflect this, the stable store must treat a modification to any part of a block as a modification of the entire block.

The stable store operates within a virtual address space that is in a one-to-one mapping with the non-volatile store. Hence, the virtual address space can be divided into pages that correspond to blocks of the non-volatile storage. The *page size* procedure returns the size of a virtual page, thereby indicating the smallest unit of stable store that may be modified by an update. An implementation of the stable storage layer can operate with virtual pages that correspond to multiple blocks of non-volatile storage but, not with pages that are smaller than a single block.

The *shadow size* procedure returns the total amount of stable storage that may safely be modified before the next checkpoint. Given the size of a virtual page, it is possible to estimate the total amount of stable storage that may be modified. When an update would cause this estimate to fall to zero, a checkpoint operation must be invoked or the stable storage may fail.

The style of system constructed using the layered architecture determines the action required when the shadow store is about to be exhausted. The architecture is intended to be used where the entire state of the system is held within the store. The timing of a checkpoint will not affect the integrity of the store since any operation can be restarted after a checkpoint. Hence, when such a system is about to exhaust the shadow store, a checkpoint can be performed and the system allowed to continue.

8.1 Estimating the use of shadow store

To support a system whose entire state is held within the stable store, the architecture must be able to automatically invoke a checkpoint when the shadow store is about to be exhausted. The specification of the architecture layers requires the abstract machine to replace any cached data held outwith the non-volatile storage before initiating a checkpoint or garbage collection operation. Thus, a mechanism is required to allow the abstract machine to monitor the usage of shadow store and invoke a checkpoint as and when required.

The abstract machine does not have direct access to the stable storage layer, instead it is constrained to operate against the heap of persistent objects. The interface to the heap prevents the abstract machine discovering the implementation of the heap within the stable store. As a result, the abstract machine is unable to calculate the amount of stable storage, and therefore shadow storage, that is required to perform a particular operation such as creating an object. This is solved in the following way.

The heap of persistent objects provides the abstract machine with an interface procedure called *estimated shadow size*. The procedure is used to estimate how much data may be modified before the next checkpoint. When the abstract machine is initialised, it calls the procedure and records the estimate. Each time the abstract machine modifies an object or creates a new object, it first deducts the size of the object from the recorded estimate. If the recorded estimate falls to zero, a checkpoint operation is invoked and the estimate is recalculated.

The calculation of the estimate must ensure that the abstract machine cannot exhaust the shadow store. Thus, the total available shadow store must be scaled down to take account of the potential overheads involved in modifying an entire object or creating a new object.

Any modification to the stable store involves changing an integral number of virtual pages. Consequently, the modification of an entire object will result in the modification of every virtual page that contains any part of the object. Similarly, the creation of an object may result in the modification of every virtual page that contains part of the new object. The creation of a new object may also involve an additional overhead. For example, the heap organisation may maintain a free list that must be updated when an object is created. Performing the necessary updates to the free list may result in the modification of one or more virtual pages.

The estimate of how much data may be changed between checkpoints can be calculated as follows. Firstly, take the smallest object that can be created and calculate the maximum number of virtual pages that could contain part of the object:

max pages = **if** (size of smallest object **remainder** page size) > 1
then ((size in words - 1) **quotient** page size) + 2
else ((size in words - 1) **quotient** page size) + 1

Secondly, calculate the maximum number of virtual pages that may be modified in creating the object:

! two words are modified in unlinking a cell from a free list and the object is initialised.
pages for newobject = max pages + 2

Finally, compare the size of the smallest possible object with the maximum amount of the stable storage that may be modified when the object is created or the entire object is modified:

$$\text{ratio size to stable store} = \frac{\text{size of smallest object}}{\textit{page size} * \textbf{largerof}(\text{ max pages } \textbf{or} \text{ pages for newobject })}$$

This ratio is the smallest possible ratio of modified data to shadow store usage. If the total size of the available shadow store is scaled down by this ratio, the creation or modification of the resulting amount of data can never exhaust the shadow store:

estimated shadow size = *shadow size* * ratio size to stable store

8.2 Conclusion

The design of the layered architecture provides a mechanism that allows it to modify its behaviour to prevent the available shadow store being exhausted. This ability to dynamically configure the individual storage layers allows experiments to be conducted at very little cost. For example, a developer may wish to experiment with the relationship between the size of shadow storage and the available stable storage. Using the dynamic configuration, only the implementation of the stable storage layer would need to be modified since the behaviour of the remaining layers would automatically change to reflect the new physical configuration.

9 Conclusion

This thesis presents research into the design and implementation of architectural support for persistent object stores. The research was undertaken as part of the Persistent Information Space Architecture, PISA, project[atk87a].

The research presented may be divided into the following parts,

a) the design and implementation of the CPOMS persistent object manager for PS-algol,
b) an analysis of the practical advantages and disadvantages of the CPOMS design, and
c) the design and implementation of a layered architecture based on the results of the CPOMS analysis.

9.1 The PSPOMS

Prior to the design of the CPOMS, PS-algol was supported by a persistent object management system written in PS-algol, the PSPOMS[atk83b]. The persistent store implemented by the PSPOMS supported the same functional interface as that of the CPOMS. However, the operation of the PS-algol standard procedures *open.database* and *commit* was significantly different.

The PSPOMS employed a database locking protocol that permitted an individual database to have multiple readers or one writer but it did not allow a program to open more than one database for writing. The limit of one writeable database was introduced to simplify the task of allocating new objects to a database. Since allocating an object to a database requires the database to be modified, the single writeable database was the only one that could be allocated new objects.

The other significant difference between the operation of the PSPOMS and the CPOMS is the semantics of *commit*. In the CPOMS a *commit* will attempt to copy every changed

persistent object back to the persistent store. In addition, any newly created objects reachable from a persistent object are also copied. The effect of the CPOMS commit is that either every change made by a program is applied to the persistent store or none of them are.

In contrast, the *commit* provided by the PSPOMS only operated on objects that were held in the writeable database and were reachable from its root object. Therefore, any changes made to objects from other databases were not considered. Also, newly created objects would not be written to the persistent store unless they were reachable from the writeable database's root object. However, the test for reachability only made use of those objects that had been copied from the persistent store. As a result, a new or modified object could be considered unreachable even if an alternative path existed within the persistent store. Such objects would not be written to the persistent store.

The semantics of the PSPOMS *commit* presented certain problems to a PS-algol programmer. For example, PS-algol provides persistence as an abstraction over the physical properties of data such as which database contains which object and which objects have been copied from the persistent store. However, the effect of a *commit* could only be determined if the programmer knew in which database an object was located and which objects had been copied from the persistent store. As a direct consequence of this, a programmer was unable to determine which of a program's changes would be preserved by a *commit* and which would not.

The problems associated with the PSPOMS arose as the result of combining the programming language S-algol with previous research in developing an object store. The design of S-algol relied on a heap to provide dynamic storage allocation, thereby abstracting over some of the physical properties of a program's data. This feature of S-algol was easily extended to provide a persistent store. In contrast, the research into the development of an object store had been based on the assumption that a program would have some explicit knowledge of the physical properties of its data. As a result, the design of the PSPOMS

assumed that a program would know in which database an object was located, whereas the development of PS-algol from S-algol abstracted over this information.

9.2 The CPOMS

The CPOMS was designed to overcome the major problems of the PSPOMS in the following ways. Firstly, the locking protocol applied to databases retained the one writer or many readers for an individual database but was extended to allow a program to open any number of databases for writing. The allocation of newly created objects to databases was performed by placing a new object in the database of an object that addresses it. This feature of the CPOMS allows updates to be performed over more than one database at a time.

To complement the operation of *commit* the CPOMS also provides an error reporting facility by making all of the database operations return a result value. In the case of *commit* the return value may be a PS-algol structure containing containing three strings that describe an erroneous condition. This information permits the running PS-algol program to take some corrective action that may alleviate the problem. This provides a programmer with further control over the effect of his program on the persistent store.

The overall effect of the CPOMS design is that it is possible to determine which modifications a program may make to the persistent store without requiring an explicit knowledge of where any particular object is held.

Although the CPOMS does avoid some of the problems associated with the PSPOMS, it is far from ideal. The PS-algol databases were designed to support coarse grain sharing but are subject to two practical limitations. Firstly, the persistence abstraction provided by PS-algol attempts to hide the locality of an object. This can prevent a PS-algol programmer knowing which database contains a particular object except in specifically engineered situations. All a programmer is allowed to know is a path from a database's root object to an object.

The second limitation is a consequence of how the CPOMS opens a database. In order to ensure that every object address encountered refers to an open database, the act of opening a database also opens every other database that may contain a reachable object. This causes the other databases to become locked thereby reducing the potential for concurrent access to the persistent store.

A further restriction on the practical use of the CPOMS is imposed by the object addressing mechanism. As it is implemented, it copies objects from the persistent store into a program's heap eventually causing the heap to become full. When this occurs the standard PS-algol system fails and reports a heap overflow. This restriction has been partly overcome by an experimental CPOMS that can discard unchanged objects[spa88].

The disadvantages of the CPOMS design also affects its use for experimentation. The structure of a PS-algol/ CPOMS system requires a large amount of information to be shared by its components. For example, the different formats of persistent object must be known to the CPOMS, the PS-algol abstract machine and the PS-algol compiler. Thus, an experiment involving a new object format requires appropriate changes to be made to the CPOMS, abstract machine and compiler. Other examples of the system's vertical structure include:

a) every component of the CPOMS must provide its own recovery mechanism,
b) the compiler, abstract machine and CPOMS must agree on PS-algol's predeclared structure classes and standard procedures, and
c) the abstract machine and the CPOMS must agree on the allocation of reserved persistent store addresses.

The overall effect of the vertical structure of the CPOMS is that constructing an experiment can be a complex, time consuming and error prone task. As such, the CPOMS is not a suitable tool for the purposes of conducting experiments.

In contrast to its many disadvantages, the PS-algol/ CPOMS system has proved successful in two areas, the protection of the persistent store and the performance of the object addressing mechanism.

All programs that use the architecture must be written in PS-algol and compiled with the PS-algol compiler. Since PS-algol abstracts over an object's implementation, this guarantees that the persistent store will not be misused. As a direct consequence of this, the CPOMS does not need to implement its own protection mechanism.

The performance of the object addressing mechanism has been greatly enhanced by the particular choice of address translation rules. These rules were designed to allow local heap addresses to be used instead of persistent store addresses whenever possible. As a result, a program will only incur a significant performance overhead while it is copying objects into the local heap. Thereafter, the overheads are reduced to a series of sign tests that can distinguish a positive main memory address from a negative persistent store address.

9.3 The layered architecture

Based on the experience gained in designing and implementing the CPOMS, a new architecture was designed. Its design was motivated by the following aims:

a) to support a type secure persistent system,
b) to support cost effective experiments with concurrency, transactions and distribution and
c) to provide the architecture's protection mechanisms via a high level programming language.

A type secure persistent system is based on a stable store whose physical properties are completely hidden from a programmer. To reflect this, the new architecture enforces a clear distinction between the storage of an object and its use. This is achieved by structuring the

architecture in such a way that a program has no knowledge of the implementation of the physical store.

To support low cost experimentation, the new architecture has been designed as a set of highly modular layers. This allows individual components to be reimplemented independently of each other provided that they conform to their specified interfaces. This avoids some of the problems associated with modifying the vertical structure of the CPOMS. To meet the requirements of the first design aim, the chosen architecture layers are divided into two groups, those that support the storage of an object and those that support the use of an object.

All programs that use the total architecture are required to be compiled into the intermediate programming language PAIL and are therefore subject to PAIL's own protection mechanisms. This achieves the architecture's third design aim. The use of PAIL also has two further consequences. Firstly, PAIL was designed to support high level programming languages and does not allow a program to access an object's implementation. Therefore, the layers that support the storage of an object need not provide their own protection mechanisms. Secondly, a particular operation cannot be supported by the architecture unless it can be programmed in PAIL. This requires PAIL to provide sufficient facilities to support each desired use of the architecture.

An abstract machine is provided to separate PAIL from the architecture's storage layers. It supports a set of abstract instructions suitable for executing PAIL programs together with an implementation of any primitives PAIL requires. Although the abstract machine can access the storage layers, its abstract instructions can only access persistent objects. As a result, the abstract machine is able to isolate PAIL from the storage layers, thereby enforcing the distinction between the storage of an object and its use.

The overall structure of the resulting architecture can be divided into the following three major components, shown in Figure 9.1:

a) those layers that support the storage of objects,
b) those layers that provide PAIL and the abstract machine and
c) those layers that support the use of objects.

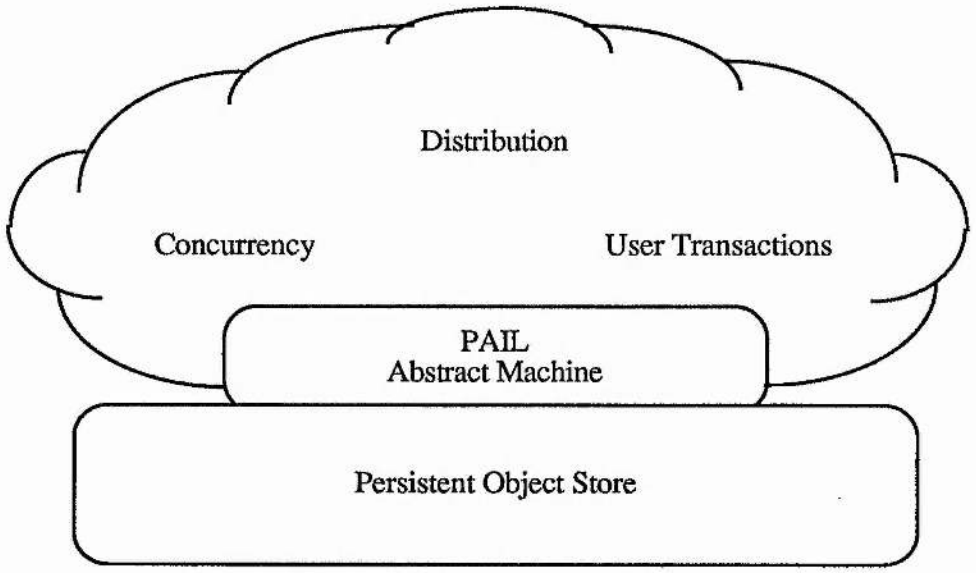


Figure 9.1. The three major architecture components.

9.4 A layered persistent object store

The combination of PAIL and the abstract machine has several important consequences for the architecture's storage layers. Firstly, the storage layers need not support concurrency, user transactions or distribution since these three mechanisms are supported by the abstract machine. In addition, the store does not need to implement its own protection mechanisms because it will not be misused. Finally, the interpretation of an object is the joint responsibility of the PAIL layer and the abstract machine. To reflect this, the storage layers provide a single object format whose interpretation is dependent on its particular use. This separation of object format from interpretation allows the storage layers to operate on an object without any knowledge of how the object is interpreted.

Since the single object format is specified as part of the interface to the heap of persistent objects, a change to this object format requires corresponding changes to be made to the abstract machine and the PAIL layer. Although this represents a degree of vertical structure, the effects of changing the object format only affect three layers. Similarly, the effects of changing the interface to any of the layers is limited to those layers that use the modified interface.

The storage layer visible to the abstract machine provides a heap of persistent objects, where each object conforms to a single object format. The heap is implemented as a set of object management procedures that organise a single contiguous stable store. To ensure that the heap is correctly used, its interface includes a set of five conventions to which PAIL and the abstract machine must conform. They are:

a) objects will only be created by the heap management procedures,
b) addresses will not be manufactured,
c) all addresses will be held in the address fields of an object,
d) all addressing is performed by indexing object addresses and
e) a reachable object will not be explicitly deleted.

These conventions ensure that objects can only be accessed by following object addresses starting from the store's root object. They also ensure that all object addresses are held in the persistent store and can be easily located. This facilitates the implementation of storage utilities such as garbage collectors. All programs using the architecture are guaranteed to conform to the conventions since they must be compiled into PAIL.

The storage layers are designed to support a stable store. To reflect this, a checkpointing mechanism is provided by the heap management procedures to simulate stability. At any point in time, the checkpointing mechanism can be used to ensure that the current state of the

system is held in non-volatile storage. Thus, a transaction mechanism can be constructed by forming a log of operations, forcing the log to be written to non-volatile storage and only then performing the desired operations.

The stable storage that supports the heap is itself supported by non-volatile storage. The purpose of the stable storage layer is to provide access to a contiguous section of non-volatile storage that is always in a self-consistent state. This is achieved via two mechanisms, a virtual addressing mechanism that views the entire non-volatile store as a contiguous virtual memory and a shadow paging mechanism that maintains duplicate copies of changed pages. To complete the layer, a checkpoint procedure is provided to establish a new self-consistent state.

The third and final storage layer provides access to all the available non-volatile storage as if it were a contiguous set of equal sized blocks. When combined with PAIL, the three storage layers implement a type secure persistent object store.

9.5 Implementation progress

At the time of writing the architecture's implementation consists of the storage layers described in Chapter 6, the abstract machine, PAIL and the programming language Napier. The implementation is sufficiently advanced to support Napier programs that operate on data held within the store, however, certain aspects of the total system have yet to be realised. For example, the dynamic storage configuration techniques described in Chapter 8 and the distribution support described in Chapter 7 have not been fully implemented.

The compilation system is currently written in PS-algol and operates in a batch oriented manner. This is in the process of being rewritten in Napier with the intention of performing all compilations within the persistent store. When this is complete, it will be possible to operate the layered architecture as a single integrated system.

The other aspects of the architecture yet to be implemented include the provision of concurrency, transaction and distribution mechanisms.

In the near future it is planned to implement the concurrency mechanism proposed by Morrison[mor87]. The implementation will require an extension to Napier with suitable support being provided within PAIL and the abstract machine. This will be followed by experiments with possible distribution mechanisms starting with the proposal described in Chapter 7. Finally, the transaction mechanisms proposed by Krablin[kra87] will be examined within the context of the layered architecture.

9.6 Future research

Once the concurrency, transaction and distribution mechanisms have been implemented the layered architecture will be complete. The architecture will then be used as the basis of further research in the following areas,

a) models of locality suitable for a distributed persistent system,
b) supporting unreliable components within a distributed persistent system,
c) developing high performance systems,
d) fine grain concurrency control and
e) models of transaction suitable for a persistent system.

It is envisaged that the most important area for further research will be that of distributed systems. At the present time the concept of orthogonal persistence has only been realised in relatively small isolated systems similar to the PS-algol/ CPOMS system. One reason for this is the practical difficulty of reconciling the persistence abstraction with distribution.

The persistence abstraction hides all the physical properties of data such as how it is stored and where it is stored. As a result, the physical distribution of data within a computer network must be completely hidden. However, if a component of a network fails the data it

contains may cease to be available. Since there is no notion of locality a program may fail unexpectedly when the data is accessed.

A potentially more serious situation could arise if a component system suffered a catastrophic failure that resulted in the loss of a checkpointed state. This could have the effect of leaving the network's persistent store in a logically inconsistent state. For example, if a complex data structure spans several systems, part of the data structure may spontaneously change state and violate any higher level consistency that a program had established between the data structure's components.

These two problems are an indication of some of the potential difficulties involved in combining the persistence abstraction with distribution. As part of the research into distributed persistence, the relationship between the ideal of orthogonal persistence and the use of inherently unreliable computer systems will be investigated.

Further research into the design of a high performance implementation is necessary to support large scale applications based on the architecture. It will then be possible to select the appropriate concurrency and transaction mechanisms to suit the architecture's particular patterns of use. The flexibility of the layered architecture is such that it can evolve to meet an individual application's requirements with respect to concurrency control and transaction mechanisms.

The initial implementation techniques to be employed in developing a high performance system will be based on the use of memory mapped files and special purpose hardware. It is intended to make use of the memory mapped files from version 4 of the SUN/UNIX system[sun88] to implement an efficient shadow paged stable store. The other proposed technique is to microcode the Rekursiv computer[bel88] to support the abstract machine and the storage layers. The resulting implementations should provide instances of the layered architecture whose performance is comparable with traditional computer architectures.

A1 Garbage collecting the main memory heap

This appendix describes the garbage collection algorithm used by the first implementation of the main memory heap, described in Section 6.2. The algorithm is an adaption of a compaction algorithm given by Morris[mor78]. The purpose of the algorithm is to slide all the accessible objects towards the bottom of the main memory heap, as shown in Figure A1.1, thereby allowing all storage allocation to be performed without any fragmentation problems. That is, all the heap's free space is available as a single unit.

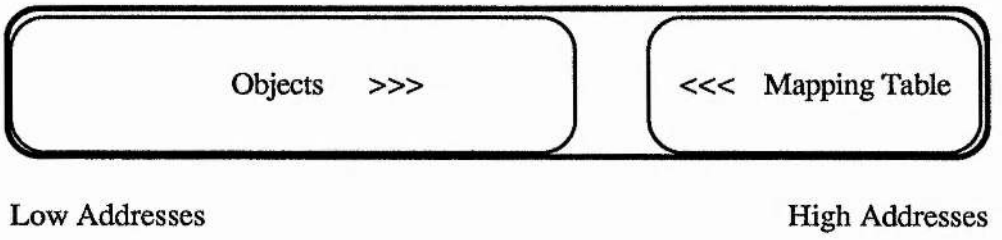


Figure A1.1. The layout of the main memory heap.

The algorithm is divided into four parts, a mark phase and three scans. The mark phase is necessary to identify those objects that should be retained by the garbage collection. An object will be retained if it is reachable from the heap's root object or if it is reachable from an object with a key. The objects with keys must be kept for two reasons. Firstly, they may have been changed since they were copied requiring the copy to be retained. Secondly, if a copied object is to be discarded any copies of its main memory address must be overwritten by the object's key. This may involve compromising one or more of the address translation rules given in Section 6.2.4. Thus, the deletion of such objects is beyond the scope of the garbage collection algorithm.

The marking algorithm operates as follows. Firstly, the KRT is scanned and every object with a key is marked. Once this has been completed the KRT becomes a list of marked objects. The marking then continues by performing a single scan of the KRT that inspects the address fields of each marked object. When the address of an unmarked object is found,

the object is marked and its address is appended to the KRT. Since the KRT has been extended, the marking scan will also inspect the newly marked object with result that the KRT may be further extended.

When the marking scan terminates, every reachable object will have been marked and its address will be present in the KRT. Since there is a single word allocated to the KRT for every object in the heap, the marking algorithm is guaranteed sufficient workspace to operate successfully. To complete the marking phase, the end of the KRT is retracted in order to remove the extra addresses appended by the marking algorithm.

Following the marking phase three scans of the heap are performed to compact the reachable objects towards the bottom of the heap. For example, Figure A1.2 shows a heap of three reachable objects and Figure A1.6 shows the heap after the compaction.

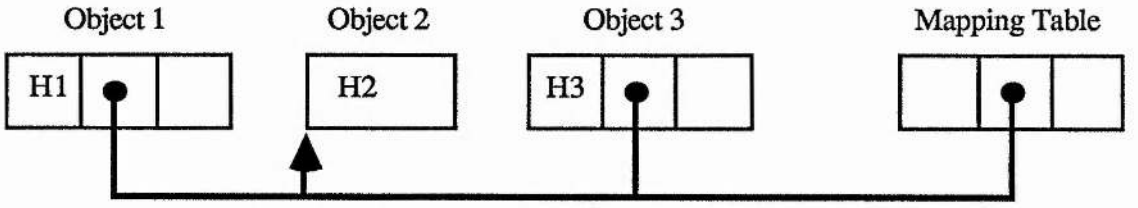


Figure A1.2. A heap of three objects.

In the first scan, every marked object has its address fields inspected. If an address field contains the address of the current object or an object at a lower address then the contents of the address field are swapped with the contents of the word it addresses. The effect of this scan is to construct a chain of backward references to an object. The last entry in the chain contains the object's header word. In our example heap, this involves the address field in object 3 being swapped with the header word of object 2, as shown in Figure A1.3.

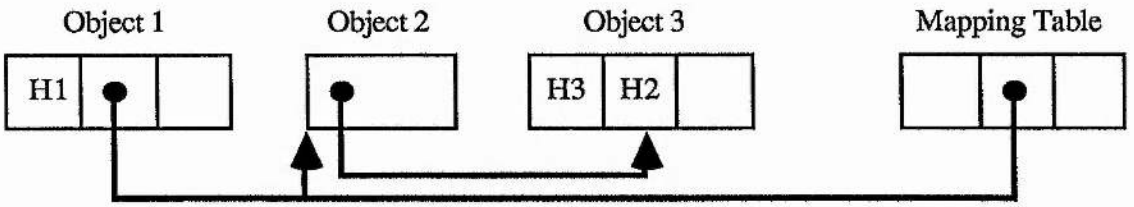


Figure A1.3. The heap after scan 1.

In the next scan, each word in the KRT that contains an address is swapped with the contents of the word it addresses. Thus, in our example heap, the KRT entry for object 2 is swapped with the first word of object 2. The effect of this scan phase is to add an object's KRT entry to its chain of backward references, as shown in Figure A1.4.

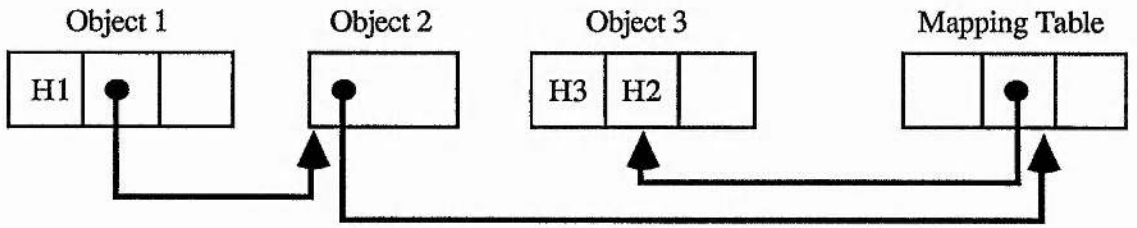


Figure A1.4. The heap after scan 2.

The final scan combines the act of compacting the reachable objects with that of modifying all the copies of an object's main memory address. The first word of each object is inspected to see if it is an object header. An object header has its most significant bit set making it a negative value.

When an object header is not present, the first word will be the first link in a chain of references to all the words that contained the object's address. At this point the final position of the object will be known. Therefore, each word on the chain can be overwritten by the object's new address and the object's header can be replaced. The object is then copied to its new position in the heap.

The final operation on an object is to inspect its address fields. Any address fields that contain the address of an object with a higher address, are swapped with the word they address. For example, the address field of object 1 would be swapped with the contents of the first word of object 2, the result is shown in Figure A1.5. The effect of this swapping is to extend an object's chain of backward references to include forward references. Thus, when an object is finally inspected by the third scan it will have a complete record of all references to it.

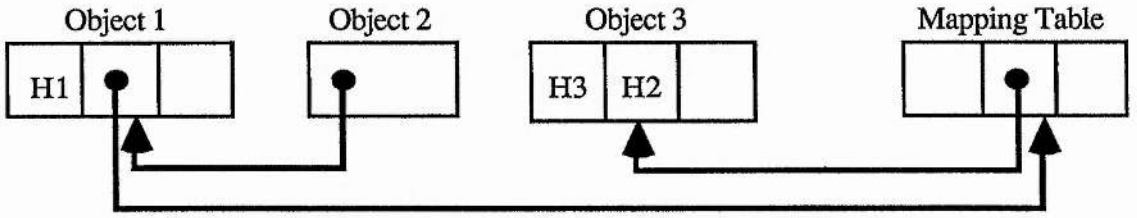


Figure A1.5. The heap after traversing object 1.

Since a KRT entry will be included in its object's chain of references, the third scan will cause the KRT entry to be overwritten by its object's new address. Hence, the third scan will result in every object address being updated correctly without the need to perform a further explicit scan of the KRT. The garbage collection is then complete and the final state of the heap is as shown in Figure A1.6.

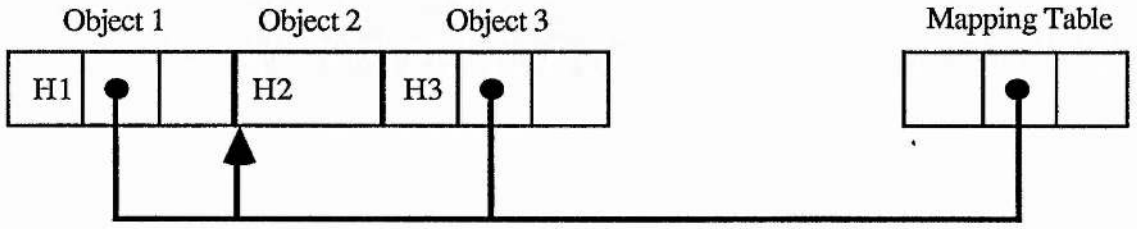


Figure A1.6. The heap after scan 3.

References

[ada83] The Programming Language Ada, Reference Manual, American Standards Institute, Inc., ANSI/MIL-STD-1815A-1983. Springer Verlag Lecture notes in Computer Science, vol. 155, 1983.

[alb85a] Albano A., Cardelli L. & Orsini R. Galilieo: A Strongly Typed, Interactive Conceptual Language. ACM Transactions on Database Systems, vol. 10, no. 2, 1985, pp230-260.

[alb85b] Albano A., Giannotti F., Orsini R. & Pedreschi D. The Type System of Galileo. Proc. First Appin Workshop on Persistence and Data Types. Universities of Glasgow and St Andrews PPRR-16, Scotland, 1985, pp175-196.

[ans78] American National Standard Programming Language Fortran, ANSI X3.9-1978. American National Standards Institute, New York, 1978.

[app86a] Inside Macintosh. Apple Computer Inc. Addison Wesley, 1986.

[app86b] Apple Numerics Manual. Apple Computer Inc. Addison Wesley, 1986.

[atk78] Atkinson M.P. Programming Languages and Databases. Proc. Fourth IEEE International Conference on Very Large Databases, 1978, pp408-419.

[atk82] Atkinson M.P., Chisholm K.J. & Cockshott W.P. CMS - A Chunk Management System. Software Practice and Experience, vol. 13, no. 3, 1983, pp259-272.

[atk83a] Atkinson M.P. Bailey P.J., Chisholm K.J. Cockshott W.P. & Morrison R. An Approach to Persistent Programming. The Computer Journal, vol. 26, no. 4, 1983, pp360-365.

[atk83b] Atkinson M.P., Bailey P.J., Cockshott W.P., Chisholm K.J. & Morrison R. The Persistent Object Management System. Universities of Glasgow and St Andrews PPRR-1, Scotland, 1983.

[atk83c] Atkinson M.P., Bailey P.J., Cockshott W.P., Chisholm K.J. & Morrison R. PS-algol Papers: A Collection of Related Papers on PS-algol. Universities of Glasgow and St Andrews PPRR-2, Scotland, 1983.

[atk87a] Atkinson M.P., Morrison R. & Pratten G.D. Persistent Information Architectures. Universities of Glasgow and St Andrews PPRR-36, Scotland, 1987.

[atk87b] Atkinson M.P., Lucking J.R., Morrison R. & Pratten G.D. Persistent Information Space Architecture - PISA Club Rules. Universities of Glasgow and St Andrews PPRR-47, Scotland, 1987.

[bab81] Babaoglu O. & Joy W.N. Converting a Swap Based System to do Paging in an Architecture Lacking Page Referenced Bits. Proc. Eighth International Symposium on Operating System Principles, ACM SIGOPS, vol. 15, no. 5, 1981, pp78-86.

[bel88] Beloff B., McIntyre D. & Drummond B. Rekursiv Hardware. Linn Smart Computing Ltd, 1988.

[bro85] Brown A.L. & Cockshott W.P. The CPOMS Persistent Object Management System. Universities of Glasgow and St.Andrews PPRR-13, Scotland, 1985.

[bro86] Brown A.L. & Dearle A. Implementation issues in Persistent Graphics. University Computing, vol. 8, no. 2, 1986, pp101-108.

[bro88a] Brown A.L., Carrick R., Connor R.C.H., Dearle A. & Morrison R. The Persistent Abstract Machine. Universities of Glasgow and St.Andrews PPRR-59, Scotland, 1988.

[bro88b] Brown A.L. The St.Andrews Disk Garbage Collector - STAG, University of St.Andrews, Scotland, 1988, In preparation.

[buc78] Buckle J.K. The ICL 2900 Series, Macmillan Computer Science Series, Macmillan, 1978.

[cam86] Campin J. & Atkinson M.P. A Persistent Store Garbage Collector with Statistical Facilities. Universities of Glasgow and St.Andrews PPRR-23, Scotland, 1986.

[car87] Cardelli L. Amber. AT&T Bell Laboratories, Murray Hill, New Jersey, 1987.

[cha78] Challis M.P. Data Consistency and Integrity in a Multi-User Environment. Databases: Improving Usability and Responsiveness, Academic Press, 1978.

[cod81] Cody W.J. Analysis of Proposals for the Floating Point Standard. IEEE Computer, March 1981, pp63-69.

[coh76] Cohen E., Corwin B., Jefferson D., Lane T., Levin R., Newcomer J., Pollack F. & Wulf B. Hydra: Basic Kernel Reference Manual. Department of Computer Science, Carnegie-Mellon University, November 1976.

[coh81] Cohen J. Garbage Collection of Linked Data Structures. ACM Computing Surveys, vol. 13, no. 3, 1981, pp341-367.

[coo81] Coonen J.T. Underflow and the Denormalised Numbers. IEEE Computer, March 1981, pp75-87.

[cos74] Cosserat D.C. A Data Model Based on the Capability Protection Mechanism. International Workshop on Protection in Operating Systems, IRIA, Rocquencourt, August 1974, pp35-53.

[dal68] Daley R.C. & Dennis B.D. Virtual Memory, Processes and Sharing in MULTICS. Comm. ACM vol. 11, no. 5, 1968, pp306-312.

[dea87] Dearle A. A Persistent Architecture Intermediate Language. Universities of Glasgow and St.Andrews PPRR-35, Scotland, 1987.

[dea88] Dearle A. (Ph.D. Thesis) On the Construction of Persistent Programming Environments. Universities of Glasgow and St.Andrews PPRR-65, Scotland, 1988.

[dec83] VAX11 Architecture Handbook, Digital Equipment Corporation, 1983.

[den70] Denning P.J. Virtual Memory. ACM Computing Surveys, vol. 2, no. 3, 1970, pp153-189.

[eng74] England D.M. Capability Concept Mechanisms and Structure in System 250. International Workshop on Protection in Operating Systems, IRIA, Rocquencourt, August 1974, pp63-82.

[fab73] Fabry R.S. The Case for Capability Based Computers. ACM Fourth Symposium on Operating System Principles, Yorktown Heights, October 1973, pp120.

[fen69] Fenichel R.R. & Yochelson J.C. A Lisp Garbage Collector for Virtual Memory Computer Systems. Comm. ACM vol. 12, no. 11, 1969, pp611-612.

[fot61] Fotheringham J. Dynamic Storage Allocation in the Atlas Computer Including an Automatic Use of Backing Store. Comm. ACM, vol. 4, no. 10, 1961, pp435-436.

[gol83] Goldberg A. & Robson D. Smalltalk-80: The language and its Implementation. Addison Wesley, 1983.

[gra81] Gray J., McJones P., Blasgen M., Lindsay B., Lorie R., Price T., Putzolu F. & Traiger I. The Recovery Manager of the System R Database Manager. ACM Computing Surveys, vol. 13, no. 2, June 1981, pp223-242.

[hol81] Holler E. Multiple Copy Update. Springer Verlag Lecture Notes in Computer Science, Distributed Systems - Architecture & Implementation, vol. 105, 1981, pp284-307.

[hou81] Hough D. Applications of the Proposed IEEE 754 Standard for Floating Point Arithmetic. IEEE Computer, March 1981, pp70-74.

[ibm78a] IBM Report on the contents of a sample of programs surveyed. IBM Research Centre San Jose, California, 1978.

[ibm78b] IBM System 38; Technical Developments. IBM General Systems Division, Pub. G580-0237, 1980.

[icl83] Introduction to PERQ. International Computers Ltd. RP10103, 1983.

[icl85] ICL VME: System Management (Series 39), International Computers Ltd. R00466/01, 1985.

[iee81] A Proposed Standard for Binary Floating Point Arithmetic, Draft 8.0 of IEEE Task P754. IEEE Computer, March 1981, pp51-62.

[ker78] Kernighan B.W. & Ritchie D.M. The C programming language. Prentice-Hall, 1978.

[kil62] Kilburn T., Edwards D., Lanigan M. & Sumner F. One Level Storage System. IEEE Transactions EC-11, no. 2, 1962.

[kno65] Knowlton K. A Fast Storage Allocator. Comm. ACM, vol 8, 1965, pp623-625.

[koh81] Kohler W.H. A Survey of Techniques for Synchronisation and Recovery in Decentralised Computer Systems. ACM Computing Surveys, vol. 13, no. 2, 1981, pp149-183.

[kra85] Krablin G.L. Building Flexible Multilevel Transactions in a Distributed Persistent Environment. Proc. First Appin Workshop on Persistence and Data Types, August 1985. Universities of Glasgow and St.Andrews PPRR-16, Scotland, 1985, pp83-106.

[kra87] Krablin G.L. Transactions and Concurrency. Universities of Glasgow and St.Andrews, PPRR-46, Scotland, 1987.

[lam76] Lampson B. & Sturgis H. Crash Recovery in a Distributed Data Storage System. Zerox Palo Alto Research Centers,1976.

[lis81] Liskov B.H CLU: Reference Manual. Springer Verlag Lecture Notes in Computer Science, vol. 114, 1981.

[lis84] Liskov B.H. Refinement - From Specification to Implementation, The Argus Language and System. Lecture Notes for the Advanced Course on Distributed Systems - Methods and Tools for Specification, Institute for Informatics, Technical University of Munich, 1984.

[lob87] Loboz Z. PS-algol Abstract Machine Monitoring. Universities of Glasgow and St Andrews PPRR-37, Scotland, 1987.

[lor73] Lorie A.L. Physical Integrity in a Large Segmented Database, ACM Transactions on Database Systems, vol. 2, no. 1, 1977, pp91-104.

[mcc60] McCarthy J. Recursive Functions of Symbolic Expressions and their Computation by Machine. Comm. ACM, vol. 3, no. 4, 1960, pp184-195.

[mil85] Milner R. The Standard ML Core Language. Polymorphism, vol. 2, no. 2, 1985.

[mor78] Morris F.L. A time and space efficient garbage collection algorithm. Comm. ACM vol. 21, no. 8, 1978, pp662-665.

[mor82] Morrison R. S-algol: A Simple Algol. BCS Computer Bulletin Series II, no. 31, March 1982, pp17,20.

[mor87] Morrison R., Brown A.L., Carrick R., Connor R. & Dearle A. Polymorphic Persistent Processes. Universities of Glasgow and St Andrews PPRR-39, Scotland, 1987.

[mor88] Morrison R., Brown A.L., Carrick R., Connor R. & Dearle A. The Napier Reference Manual. University of St.Andrews, Scotland, 1988, In preparation.

[mot85] MC68881 Floating Point Coprocessor User's Manual. Prentice-Hall, 1985.

[mye84] Myers S.E. RPG II and RPG III with Buisness Applications. Reston Publishing Co., Reston VA, USA, 1984.

[nau63] Naur P. et al. Revised report on the algorithmic language Algol 60. Comm. ACM vol. 6, no. 1, 1963, pp1-17.

[nee74] Needham R.M. & Walker R.D. Protection and Process Management in the CAP Computer. International Workshop on Protection in Operating Systems, IRIA, Rocquencourt, August 1974, pp155-160.

[org73] Organick E.I., Computer System Organisation: The B5700/B6700 Series, Academic Press, New York, 1973.

[pat81] Patterson D.A. & Sequin C.H., RISC 1: A Reduced Instruction Set VLSI Computer. Proc. Eighth Annual Symposium on Computer Architecture, ACM SIGARCH Computer Architecture News, vol. 9, no. 3, 1981, pp443-457.

[pat82] Patterson D.A. & Sequin C.H., RISC Assessment: A High Level Language Experiment. Proc. Nineth Annual Symposium on Computer Architecture, ACM SIGARCH Computer Architecture News, vol. 10, no. 3, 1981, pp3.

[pay80] Payne M.H. & Bhandarkar D. VAX Floating Point: A Solid Foundation for Numerical Computation. ACM SIGARCH Computer Architecture News, vol. 8, no. 4, 1980, pp22-33.

[per87] Perry N. Hope+. Imperial College Internal Report IC/FPR/LANG/2.5.1/7, 1987.

[pet83] Peterson J.L. & Silberschatz A. Operating System Concepts. Addison Wesley, 1983.

[psa85] PS-algol Abstract Machine Manual. University of Glasgow and St Andrews PPRR11-85, Scotland, 1985.

[psa88] The PS-algol Reference Manual fifth edition. Universities of Glasgow and St.Andrews PPRR-12, Scotland, 1988.

[ran69] Randell B. A Note on Storage Fragmentation and Program Segmentation. Comm. ACM, vol. 12, no. 7, 1969, pp365-369.

[rey79] Reynolds D.N. & Henry G.G. The IBM System 38. Datamation, 141-3, August 1979.

[ric80] Richards M. & Streven C.W. BCPL - The Language and its Compiler. Cambridge University Press, 1980.

[rit74] Ritchie D.M. & Thompson K. The UNIX Time Sharing System. Comm. ACM, vol. 17, no. 7, 1974, pp365-375.

[ros83] Ross G.D.M. (Ph.D. Thesis) Virtual Files: A Framework for Experimental Design. University of Edinburgh, 1983.

[ros85] Rosenberg J. & Abramson D.A. MONADS-PC: A Capability Based Workstation to Support Software Engineering. Proc. Eighteenth Annual Conference on System Sciences, Honolulu, Hawaii, 1985.

[ros87] Rosenberg J. & Keedy J.L. Object Management and Addressing in the Monads Architecture. Proc. Workshop on Persistent Object Systems: Their Design Implementation and Use, Universities of Glasgow and St.Andrews, PPRR-44, Scotland, 1987, pp114-133.

[sam62] Sammet J. Basic Elements of Cobol 61. Comm. ACM, vol. 5, no. 5, 1962, pp237-253.

[sch72] Schroeder M.D. & Saltzer J.H. A Hardware Architecture for Implementing Protection Rings. Comm. ACM, vol 15, no. 2, 1972, pp157-170.

[sch84] Schneider F. Refinement - From Specification to Implementation, Paradigms for Distributed Programs. Lecture Notes for the Advanced Course on Distributed Systems - Methods and Tools for Specification, Institute for Informatics, Technical University of Munich, 1984.

[sny79] Snyder A. (Ph.D. Thesis) A Machine Architecture to Support an Object Oriented Language. MIT/LCS/TR-209, Massachusetts Institute of Technology, Laboratory of Computer Science, Cambridge, Massachusetts, 1979.

[spa88] Sparks D.J. An Improvement to the PS-algol Garbage Collection Algorithm on the SUN. STC Technology Ltd., Copthal House, Newcastle-u-lyme, 1987.

[str69] Struble G. Assembler Language Programming: The IBM System 360. Addison Wesley, 1969.

[svo84] Svobodova L. File Servers for Network Based Distributed Systems. ACM Computing Surveys, vol. 16, no. 2, 1970, pp353-398.

[sun85] The UNIX System: A Sun Technical Report. Sun microsystems Inc., 1985.

[sun86a] SUN-3 Architecture: A Sun Technical Report. Sun microsystems Inc., 1986.

[sun86b] Sun Floating Point Accelerator User's Manual. Sun Microsystems Inc., Pub. 800-1378-02, 1986.

[sun86c] Floating Point Programmer's Guide for the Sun Workstation. Sun Microsystems Inc., Pub. 800-1552-1300, 1986.

[sun88] System Services Overview. Sun Microsystems Inc., Pub. 800-1753-10, 1988.

[tan85] Tanenbaum A.S. & VanRenesse R. Distributed Operating Systems. ACM Computing Surveys, vol. 17, no. 4, 1985, pp419-470.

[tei78] Teitelman W., Goodwin J.W., Hartley A.K., Lewis D.C., Vittal J.J., Yonke M.D., Bobrow D.G., Kaplan R.M., Masinter L.M. & Sheil B.A. Interlisp Reference Manual. Xerox, Palo Alto Research Centers, California, 1978.

[thak86] Thakkar S.S. & Knowles A.E. A High Performance Memory Management Scheme. Computer, May, 1986, pp8-20.

[that86] Thatte S.M. Persistent Memory: A Storage Architecture for Object Oriented Database Systems. Proc. ACM/IEEE 1986 International Workshop on Object Oriented Database Systems, Pacific Grove, CA, September 1986, pp148-159.

[tho78] Thompson K. UNIX Implementation. UNIX Time Sharing System, The Bell Systems Technical Journal, vol. 57, no. 6, part 2, 1978, pp1931-1946.

[tur79] Turner, D.A. SASL language manual. University of St.Andrews CS/79/3, Scotland, 1979.

[tur82] Turner D.A. Recursion Equations as a Programming Language. Functional Programming and its Applications: An Advanced Course. Cambridge University Press, 1982.

[ung84] Unger D. Generation Scavenging: A Non-Disruptive High Performance Storage Reclamation Algorithm. ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Pittsburgh, April 1984.

[van69] van Wijngarden, A. et al. Report on the algorithmic language Algol 68. Numerische Mathematik vol. 14,no. 1, 1969, pp79-218.

[wil82] Wilkes M.V. Hardware Support for Memory Protection: Capability Implementations. Symposium on Architectural Support for Programming Languages & Operating Systems, ACM SIGARCH Computer Architecture News, vol. 10, no. 2, 1982, pp108-116.

[wir73] Wirth N. The programming language Pascal. Acta Informatica vol. 1, no. 1, 1973, pp35-63.

[wul74] Wulf W.A. et al. Hydra: The Kernel of a Multiprocessor Operating System. Comm. ACM vol: 17, no. 6, 1974, pp337-345.