A Layered Persistent Architecture for Napier88

A.L. Brown, R. Morrison, D.S. Munro
University of St.Andrews

A. Dearle
University of Adelaide

J. Rosenberg
University of Newcastle

ABSTRACT

In recent years a range of single programming language systems have been developed that are supported by a persistent store. Examples of such systems include Argus, Galileo, PS-algol and Smalltalk. Although each of these systems is based on a subtly different concept of persistence a common approach is to utilise a layered architecture. This paper presents the design of one such layered architecture that can be used to support a persistent object store where the protection is enforced by a high level type system. The architecture has been used to construct the persistent programming system for Napier88 and is powerful enough to support languages with similar type systems.

1. INTRODUCTION

In recent years a range of single programming language systems have been developed that are supported by a persistent store [2, 4, 6, 8, 28, 29]. Examples of such systems include Argus [17], Galileo [1], PS-algol [25] and Smalltalk [13]. Although each of these systems is based on a subtly different concept of persistence a common approach is to utilise a layered architecture. This paper presents the design of one such layered architecture that can be used to support a persistent object store where the protection is enforced by a high level type system. The architecture has been used to construct the persistent programming system for Napier88 [22] and is powerful enough to support languages with similar type systems, for example Galileo, Hope$^+$ [24] and Staple [20].

The architecture is able to support programming languages that utilise the concept of orthogonal persistence. Orthogonal persistence requires the persistence abstraction to be applicable to all data types without regard to their lifetimes or patterns of use. That is, all data in a system may be manipulated independently of its physical location, size, storage format, storage media or any other physical property it may exhibit [3].

A persistent store that supports orthogonal persistence has certain perceived properties. For example, since the storage format of data is hidden, a persistent store may be viewed as a uniform store. Its size is also conceptually unbounded since the physical properties of the

storage media are hidden. Furthermore, failures are hidden with the result that the store must appear failure free.

In practice it is not possible to build a store of unbounded size or one that is failure free. However, a wide range of techniques is available that may be used to simulate the properties of unbounded size and absolute stability. In section 2 we will discuss some of these techniques and distinguish the architectural mechanisms required, namely an addressing mechanism, a storage management mechanism and a stability mechanism. The composition of a persistent store will then be described in terms of architectural layers that provide the required architectural mechanisms.

An important feature of a persistent system is that all data within the system is subject to the protection mechanisms required by the programming languages that manipulate it. Consequently, the design of the system architecture must accommodate any interactions between the different protection mechanisms that may be applied to shared data. In section 3 we discuss some possible protection mechanisms.

Finally, we describe a layered architecture for a persistent system composed from an appropriate selection of the protection and storage mechanisms. The resultant architecture, that for Napier88, has been implemented on conventional hardware and makes use of a high level protection mechanism.

The strength of the layered architecture is that it is flexible and allows a high degree of reuse without compromising efficiency. The architecture is generic in that layers may be replaced to expedite experimentation. Thus, many versions of the layers may exist as a set of tools and these may be composed, subject to the constraints of the layer interfaces, to yield an instance of the persistent architecture. This genericity, based on plug-in tool sets, allows a version of the architecture to be appropriately specialised to a particular implementation of a language.

It is also intended that each of the layers, or groups of layers, may be reused as tools in other systems. Indeed, this is exactly what has happened in the implementation of the persistent store for the language Staple.

Finally, in any implementation the layers may be virtual layers. The compiler may, for efficiency reasons, wish to avoid mapping through the interfaces. This may be achieved for any combination of the layers. Alternatively, as in the case of the Rekursiv [5], hardware can be used to implement a layer.

2. PERSISTENT STORES OF UNBOUNDED SIZE

As described above, a store that supports orthogonal persistence has certain perceived attributes including uniformity, unbounded size and absolute stability. We shall now describe a number of techniques that may be used to simulate a uniform stable store of unbounded size. As part of the discussion we shall consider the issues of addressing the object store, managing the object store and making the object store stable.

2.1. Addressing the Object Store

The first issue we will consider is how the store may be addressed. There are several levels of addressing that may be present in a computer system ranging from the symbolic addresses used by a programmer, to the logical addresses used by an instruction set, to the physical addresses that must be used by the hardware.

Since a persistent store appears uniform, a single addressing mechanism is required at the external interface in order to provide an appropriate level of abstraction over the entire store. Although the store is addressed by this single mechanism, there may be several more primitive addressing mechanisms that support it as well as several higher level mechanisms that are mapped onto the store interface. For example, the store may be viewed as an object space supported by one or more mapping tables that record the physical location of each object. In this case access to an object is achieved using a lower level addressing mechanism that may be different for each kind of physical storage in which an object may reside.

In the following sections we show three different levels of addressing abstraction that may be used to provide a uniform store. In practice a system may support multiple levels of addressing where each level corresponds to one or more of these three abstractions, in any combination.

2.1.1. Symbolic Addressing

A persistent store may be addressed purely in terms of symbolic addresses. At this level of abstraction the name of an object would be mapped onto a lower level address and a second mapping table would map a field name onto a location within the object. The result of the two mappings can then be combined to form the address of the desired data within the underlying storage. This is illustrated in Figure 1.

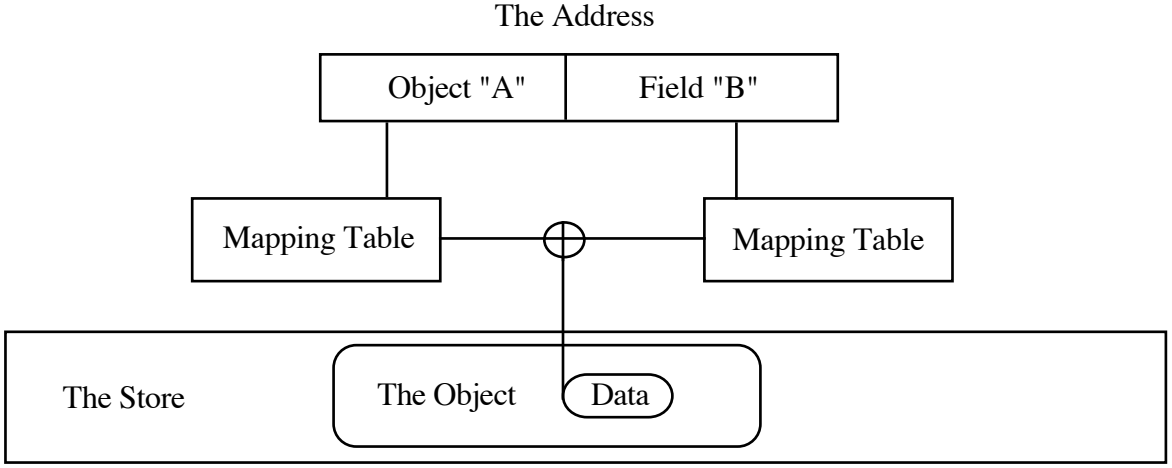

Figure 1: Addressing a Field "B" of an Object "A"

The main advantage of this uniform addressing mechanism is that it may abstract over many different physical storage mediums or lower level addressing mechanisms and it imposes no limits on the amount of storage that can be addressed. Furthermore, the dynamic name

resolution allows a program to operate over any objects that contain data with the required field names.

One disadvantage of this abstraction is that the dynamic resolution of names may be inefficient, particularly if no restrictions are placed on the length of symbolic names. This disadvantage may be alleviated by only performing the address mapping once and thereafter using the lower level address. The optimisation may not always be appropriate since it implies the preservation of a binding from the symbolic name to the lower level address. An example of a system that utilises dynamic name resolution with this optimisation is the Multics system [12].

2.1.2. Object Numbers

An alternative to symbolic addressing is to view the object store as an object space where each object is identified by a number and each field of an object is identified by an offset into the object. Thus, an address consists of two components, an object number and an offset. These may be provided as a single partitioned integer or as separate integers depending on the implementation [5, 12]. This abstraction relies on a mapping table that maps object numbers to lower level storage addresses and is illustrated in Figure 2.

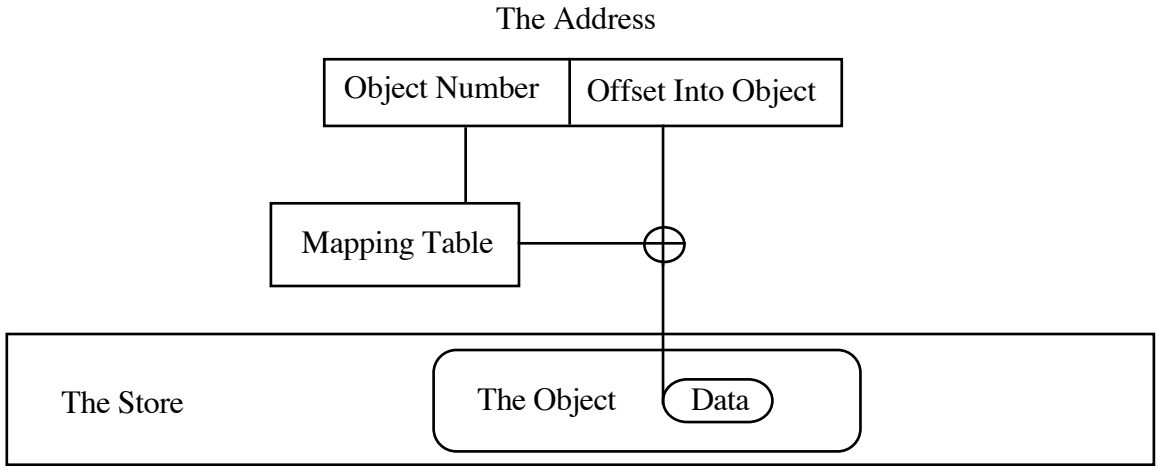

Figure 2: Addressing Data Within an Object Using a Partitioned Address

A major advantage of this addressing abstraction is that the decoding of the object address and the offset within the object may be efficiently performed in hardware. The partitioning of the address space also supports the dynamic growth and shrinkage of objects, up to the maximum length that can be addressed via an offset. Thus stack and file objects may be conveniently modelled using this approach. However, there are two potential disadvantages.

Firstly, the fixed partitioning of the address space imposes a fixed relationship between the maximum number of objects that can be created and maximum size of objects that may be created. Depending on the chosen partitioning, this ranges from a few large objects to a large number of small objects. In the absence of an additional mechanism to concatenate objects [6, 9] or the use of large addresses, the abstraction may be unable to cope with a combination of a few large objects with large numbers of small objects.

The second disadvantage is that the choice of partition size may result in object numbers being exhausted before the store is filled. This problem may be overcome by using larger addresses with any increased overheads in storage or address translation being minimised by the appropriate use of contextual addressing.

An example of a system that uses this approach is the Rekursiv. The hardware of the Rekursiv supports the efficient mapping of object numbers to physical addresses and the automatic caching of the first word of an addressed object. The Burroughs B5700/6700 series [23] is another example.

2.1.3. Virtual Addresses

The addressing of an object store may also be performed by viewing the entire system storage, in all its physical forms, as a flat virtual store and providing a higher level architecture to support an object view of this virtual store. This level of abstraction is illustrated in Figure 3. There is only one address space at this level of abstraction and it may be supported by any one of a number of well known techniques. For example, a paged virtual memory mechanism could be used based on conventional hardware.

The advantages of this scheme are that no mapping table is necessary to locate an object within the storage, locations within an object can be directly addressed, conventional hardware can be employed, alternative storage organisations can be implemented without affecting the addressing mechanism and the address space need not be larger than the available physical storage.

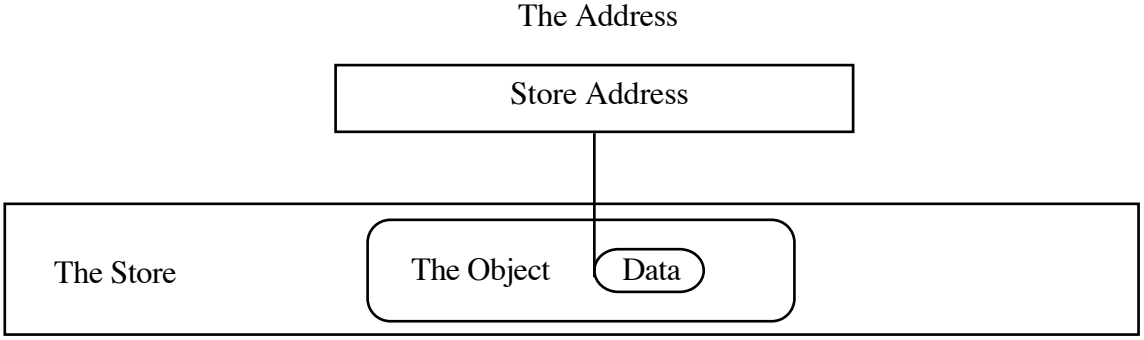

Figure 3: Addressing Data Within an Object Using a Direct Address

A disadvantage of this level of abstraction is that objects may be required to change address if the higher level architecture reorganises the mapping of objects onto virtual storage. Although a similar problem may arise in higher level addressing mechanisms, the higher level mechanisms abstract over the mapping of objects to storage thereby allowing an object to be relocated without altering its logical address.

The MONADS-PC [26] is an example of a system that utilises this level of address abstraction within its hierarchy of addressing mechanisms. MONADS supports a very large virtual address space the organisation of which is the responsibility of a higher level architecture.

2.2. Managing the Object Store

Each of the above addressing abstractions require a mechanism to organise the available storage into objects. Addressing abstractions based on symbolic names or object numbers view the organisation of the store into objects as a function of a lower level architecture whereas, the virtual addressing abstraction assumes that objects are provided by a higher level architecture. In each case the addressing and organisation of the object store may be viewed as distinct architectural layers.

There is a wide variety of techniques that may be employed to organise a store into objects. Many of these techniques have been developed for use with programming languages that support objects whose lifetimes may be independent of the procedure activations that create them. Thus, each technique has had to address the problems of dynamic storage allocation, storage reclamation and any associated fragmentation.

A store organisation can be designed to suit a particular application. However, the effectiveness of a particular choice of store organisation is dependent on both the scale of the store and the manner in which it is used. This may not be known at the design stage of a general purpose system.

2.2.1. Simulating Unbounded Size

The simulation of unbounded size involves managing an object store in such a way that, conceptually, it is always possible to create more objects. However, since a physical store is of finite size it is necessary to reuse the storage allocated to objects that are no longer required. This may be achieved either by explicitly deleting objects or performing some form of automatic garbage collection. A further consideration is whether or not the addresses of deleted objects may be reused.

The explicit deletion of objects is employed by systems that can statically determine the lifetime of data. For example, if first class procedures are not supported, the storage allocated to an activation record can be recovered when a procedure call returns. Explicit deletion of objects is also used by programming languages such as C [15] and Pascal [31].

A wide range of garbage collection algorithms has been developed and integrated with the storage allocation mechanisms of different store organisations [11]. Each of these algorithms and their host stores are designed to support a particular pattern of use or scale of data. For example, the garbage collector employed in the S-algol [21] heap storage uses a single list of free storage [19]. Since S-algol does not create objects at a very high rate the cost of object creation and garbage collection only has a small effect on overall system performance. In contrast, systems supporting first class procedures may make intensive use of a heap. Such systems require the support of sophisticated techniques, for example the buddy system [16] or generation scavenging [30], that minimise the overheads involved in storage allocation and garbage collection.

The deletion of an object requires that all references to an object are invalidated. This can be achieved by searching the store for all references to the object and removing them. Alternatively, indirect addressing could be used and the address mapping for the object invalidated. This option requires that addresses not be reused so that the mappings remain

invalid. In turn, this requires the number of available object addresses to be so large that they can never be exhausted. Examples of systems that support this technique include Hydra [10] and MONADS.

2.3. Simulating 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 wholesale 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. The techniques for recovering from hard failures range from taking complete dumps on removable media to maintaining multiple on-line copies. These techniques are out with the scope of this paper and are discussed elsewhere [8]. For the purposes of this paper we shall only consider techniques that allow the simulation of stability with respect to soft failures.

2.3.1. Soft failures

A soft failure may occur during a series of updates thereby preventing a logical operation from completing. As a result, the data held in the store may 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, a modification either completes or it is totally undone. One 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 mechanisms provided by a system may be extremely varied. For example, consider a traditional database system such as IBM's System R [14]. 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 storage 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 [18]. 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 persistent object store can adopt a much simpler transaction mechanism. Since stability is an orthogonal property of the data within a persistent store, a simple checkpointing mechanism is sufficient to ensure that the object store remains self-consistent with respect to failures. The checkpointing mechanism records incremental changes to the persistent store and may operate on individual objects or the storage in which the objects reside.

Shadow paging is used by the PS-algol/ Shrines system implemented under VAX/VMS [28]. 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 checkpointing mechanism supported by Shrines allows the new version of the persistent store to become the original in a single atomic action. A similar scheme is proposed to support stability in the MONADS machine [27].

The alternative approach, adopted by systems such as the PS-algol/ CPOMS system [6], is to record different versions of an object rather than different versions of a page. 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 between each checkpoint.

The choice between a before or after look will depend on the particular use made of a store. For example, the design of the CPOMS anticipated that updates to the persistent store would contain a large proportion of new objects. Thus, a before look was chosen since it would have to record less data than an after look. 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.

2.4. Composing a Persistent Object Store

A persistent object store may be implemented by composing a suitable selection of the techniques described above. For example, any combination of the three levels of addressing abstraction can be adopted to provide a uniform store. The selected addressing abstractions can then be combined with a storage management scheme that is able to simulate unbounded size. This composition will result in a uniform store of unbounded size. Finally, a stability mechanism may be integrated with the store that operates in terms of objects or the storage in which the objects reside. In either case stability is an orthogonal property of the store and as

such it may be viewed as a distinct architectural layer. The result of the composing the three architectural layers namely, the addressing abstraction, the storage management and the stability mechanism, is the simulation of a uniform, stable, object store of unbounded size.

Although the result of a particular implementation strategy may be to merge or otherwise integrate the three architectural layers, the functionality of the layers can still be distinguished. In the architecture to be described these distinctions are preserved by forcing the separate implementation of the architectural layers. The resulting architecture is flexible enough to allow each layer to be reimplemented independently of the others. An instance of the layered architecture may be composed from an arbitrary choice of layer implementations even if some of the layers make use of special purpose hardware. This permits cost effective experimentation with implementation techniques and the manner in which these techniques interact within the context of the architecture.

3. TYPE SECURITY

All data within a persistent object store may be manipulated without regard to its physical attributes. Hence, the data may be manipulated by any programming language supported by the system architecture. This requires the data to be subject to the protection mechanisms required by those programming languages that manipulate it. Furthermore, the protection mechanisms applied to shared data must not be able to compromise each other.

3.1. Store Level Protection

The protection mechanisms provided by a persistent system are dependent on the kinds of programming language that are supported. For example, if programming languages such as C or assembly language are supported then the protection mechanism must be applied at the storage level. These languages may arbitrarily manipulate addresses and thereby access the implementation of an object. Thus, every operation on the store must be dynamically checked to ensure that it is safe. That is, an attempted store operation must conform to a predefined set of type rules and it must not allow a program to gain unauthorised access to data. A store may be described as type secure if all the permitted operations are safe.

At the store level type security must be enforced in two ways. Firstly, since programming languages such as C may exhibit arbitrary behaviour, a mechanism is required to prevent programs manufacturing or capturing addresses that could be used to gain unauthorised access to data. Secondly, the interpretation of the accessible data must also be controlled so that programs only apply appropriate operations to the data. Systems that provide this form of store level protection are known as capability systems and require some level of hardware support.

One technique that is used to prevent addresses being manufactured is to segregate address from non-address data and to only permit certain operations on the addresses. The operations may be limited to the creation of new objects and to copying an address between address only storage areas. An alternative technique is to tag locations containing addresses and to automatically reset the tags if the locations are updated. In this way an address is invalidated if it is illegally altered. The address manipulation facilities provided by the system preserve the tags.

To complement the controlled creation of addresses capability systems may provide mechanisms to limit the propagation of addresses. For example, a limited copy access right might be used to copy an address but the copy of the address may not be copied. Another technique is to associate a key with an address. The address may be freely passed around between programs but it may only be used in conjunction with the original key. This limits the context in which an address may be used.

In addition to controlled address propagation a capability system may support the revocation of addresses. That is, an address may be invalidated and all access to the object to which it refers can be removed. This may be expensive to implement since it may require the support of indirect addressing or the ability to find all references to an object.

The control of address creation and propagation is just a special case of the controlled interpretation of data. In the non-address case this is usually of a very limited nature. For example, access to an object may be restricted to read, write or execute, without any additional constraints on how the data should be interpreted. Thus, an object containing a floating point number may be erroneously viewed as an integer without an error being detected.

A less primitive approach is to tag individual locations within an object and thereby specify the type of data each location contains. This allows simple data types such as integers and floating point numbers to be differentiated. Other supported data types may include pixels, addresses, character strings, arrays, structures and procedure closures.

The main advantages of store level protection are that it can support programming languages such as assembly language and C, it can allow arbitrary combinations of programming languages and segregate them if necessary, it can dynamically alter access to and interpretation of data, and for simple data types it can be efficiently implemented in hardware. However, on their own, store level protection mechanisms may not be able to efficiently support recursively typed data structures since the necessary run time checks can prove extremely expensive. Furthermore, once a particular mechanism has been implemented in hardware it may be very costly to alter.

Finally, in isolation, this approach to protection limits the programmer's confidence that a program is correct. That is, certain programming errors may not be detected until runtime and any programming language data types not supported by the protection mechanism may be misinterpreted resulting in erroneous program behaviour.

3.2. High Level Protection

Type security may be enforced at a higher level of abstraction by a programming language hiding the implementation of objects. In this case, a compiler may check the operations to be performed by a program to see if they conform to the type rules. This allows a program to run without the overheads of dynamic checking and may permit optimisations to be performed. For example, accessing a structured object may involve checking that the object exists, checking that the object contains the required data and finally indexing the object. If the result of the two checks can be determined by a compiler then the data may be directly addressed.

A further advantage of high level protection is that very sophisticated type systems may be used. These type systems may require expensive type checks to be performed but, the type checking need only be performed once at compile time and not each time a program accesses data of a particular type.

Not all operations may be fully checked at compile time. For example, a vector indexing operation may require a run time check to ensure that a legal index is used. In cases such as this, the compiler is able to generate additional code to dynamically check the operation. However, the dynamic checking may be simplified by removing from it any component checking that may be statically determined.

Statically determining the type correctness of a program reduces the range of potential errors that may occur at runtime. For example, any attempt to misinterpret a programming language data type will be detected at compile time. In comparison with purely dynamic checking, this increases the programmer's confidence in the correctness of a program.

Although high level protection mechanisms can provide sophisticated control over the interpretation of data they are not well suited to controlling access. One reason for this is the assumption that once access to an object has been established it is permanently available. This problem may be alleviated by extending the type rules to include data types that must be dynamically checked for availability. However, this may reduce a programmer's confidence in a program being correct in that it introduces a potential source of programming errors.

The combination of programming languages requires any shared data to have an equivalent interpretation under both type systems. If this is not possible the programming languages must be totally segregated from each other. The segregation need only be a logical structuring of an object store if high level protection is sufficient for each programming language, otherwise the segregation must be enforced by a store level protection mechanism.

High level protection may allow a type secure store to be efficiently implemented without the need for special purpose hardware. If this is the case, the protection mechanism may be altered relatively cheaply since no hardware need be modified. However, dynamic checking may be delegated to a store level protection mechanism if one is available. Finally, a high level protection mechanism enables a program's view of data to be an abstraction over the physical storage of the data.

To summarise, high level protection provides the following benefits:

- There are no unnecessary dynamic checks.
- Optimisations may be possible as a result of static checking.
- Sophisticated type systems may be supported.
- The range of potential runtime errors is reduced.
- Special purpose hardware is not required to support an efficient implementation.
- An abstract view may be imposed over the physical storage of data.

4. DEVELOPING THE LAYERS

In the preceding discussions the distinct architectural mechanisms required to support a persistent object store have been identified. They include, a uniform addressing mechanism,

a storage management mechanism, a stability mechanism and a protection mechanism. We shall now present the design of a layered architecture that provides the above mechanisms as a set of distinct architectural layers. The architecture to be described supports orthogonal persistence and has been used to implement the persistent programming language Napier88 and the functional programming language Staple.

4.1. The Basic Layers

The layered architecture has been designed with the aim of supporting cost effective experimentation with the implementation of persistence. The key to achieving this aim is the separation of the distinct architectural mechanisms into well defined layers. Thus, each architectural mechanism is provided by a distinct architectural layer that must conform to a particular specification. In this way, individual layers may be independently reimplemented without reference to the implementation of the other layers. It is also possible to merge adjacent layers provided that the interface to the top-most layer is preserved.

The architectural layering has been chosen to take advantage of the persistence abstraction by ensuring that programs are not able to discover details of how objects are stored. This divides the architecture between the architectural layers that provide the persistent object store and the those facilities that may be programmed by a supported programming language. The architectural layering is shown in Figure 4.

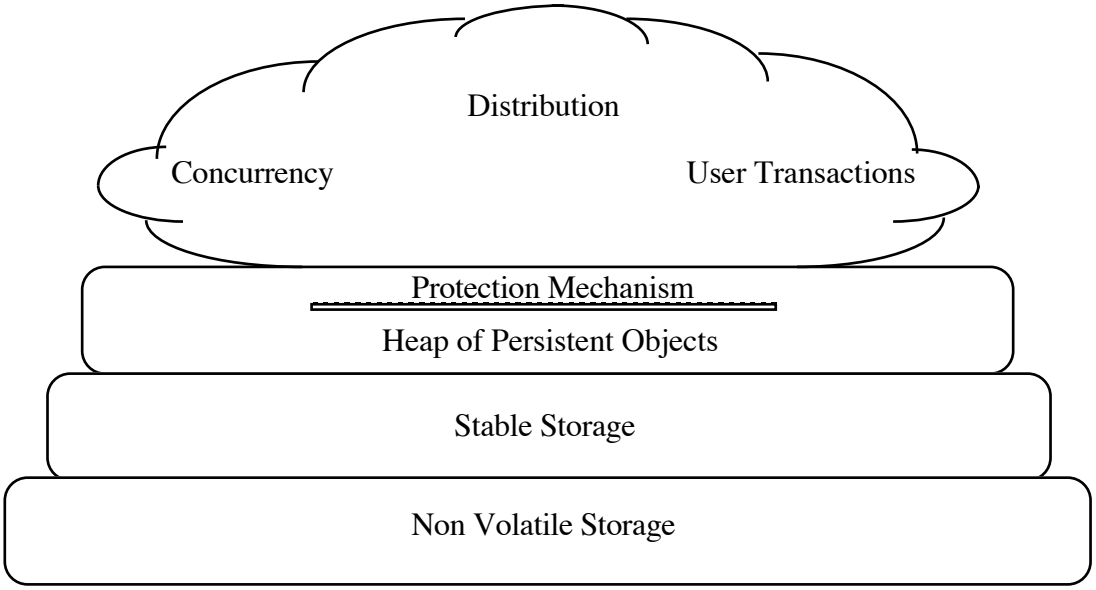

Figure 4: The Basic Architectural Layers

The division has an important consequence for the provision of concurrency, transactions and distribution. Since each of these three mechanisms are essentially modelling techniques they may be implemented by the programming language level and need not be primitive facilities provided by the persistent store. This allows experimental implementations to be constructed without the need to redesign the entire architecture. However, once a particular implementation technique has been identified as essential one or more layers of the persistent store may be reimplemented to incorporate the mechanism. If a layer interface is changed the

change is only visible to the layer immediately above thereby limiting the required reimplementation.

4.2. The Persistent Object Store

The layer of the persistent object store which is visible to the programming language level, is the heap of persistent objects shown in Figure 4. The heap layer provides a view of the persistent store that appears stable, is conceptually unbounded in size and may be uniformly addressed. All objects in the heap are reachable from a single distinguished root object and conform to a single object format that distinguishes object addresses from non-address data. The interpretation of an object is responsibility of the higher level architecture. The persistent object store does not support object formats specific to any particular programming language, thereby allowing the persistent object store to operate independently of the supported programming languages.

Within the persistent store, stability is simulated by a simple checkpointing mechanism. This mechanism is provided as part of the heap interface for two reasons. Firstly, it may be made available to the programming language level to support user level transactions. For example, a transaction may maintain a log of operations to be performed and may wish to ensure that the log is preserved in stable storage prior to performing the actual operations. Another reason for making the checkpoint explicit is that it allows the higher level architecture to cache data outwith the persistent store. When a checkpoint is required any data held in registers or other special purpose hardware is copied back to the persistent store. Thus specialised code generation techniques can be used without impacting on the implementation of the persistent store.

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 the higher level architecture must conform. They are:

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

These conventions ensure that objects can only be accessed by following object addresses starting from the root object of the persistent store. 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 that may be used to simulate the perceived property of unbounded size.

Adherence to the heap interface requires the higher level architecture to address the store in terms of indexing object addresses. However, it does not define the level of addressing abstraction employed. Thus, a particular heap implementation may treat object addresses as object numbers and perform all addressing via table lookups to determine an object's address in the stable storage. Alternatively, object addresses may be in the form of stable storage addresses and not require mapping by the heap implementation. In either case the higher

level architecture is constrained to address objects using an object address and a separate index. This corresponds to the addressing abstraction described in section 2.1.2.

The heap layer forces the higher level architecture to view the persistent object store as a uniform stable store of unbounded size. To extend this view to that of a type secure persistent object store a protection mechanism is required to ensure that the higher level architecture conforms to the specified conventions and correctly interprets the data held in the store. The architectural layering can support both high level and low level protection mechanisms.

Low level protection may be supported by encoding the appropriate checking mechanisms into the heap implementation. This may be further complemented by tagged memory locations. For example, the implementation of the architecture on the Rekursiv enforces the interface definition using a hardware address translator that only accepts object number, index pairs and by tagging addresses to prevent their unauthorised manufacture. Similar approaches to store level protection may be employed by alternative implementations of the heap layer.

High level protection may be provided by compiling all supported programming languages against a compatible type system with suitable dynamic checks being planted to accommodate those situations that cannot be statically checked. This approach allows the persistent object store to assume that all attempted operations are type correct. However, to achieve an efficient implementation without hardware support an instance of the architecture is constrained to use programming languages that make exclusive use of high level protection. Otherwise, some hardware support may be necessary to efficiently implement the dynamic checking.

The provision of a low level protection mechanism must be specified as part of the heap interface. For example, if a heap implementation does not provide a low level protection mechanism then it can only support programming languages that rely on high level protection. Thus, the heap interface must specify the supported protection mechanism to ensure that an instance of the architecture is composed from compatible layer implementations.

The architecture implementation for Napier88 relies on high level protection and requires all programs to be compiled by the one compilation system.

4.3. The Stable Store

The heap layer is directly supported by a single contiguous stable store, see Figure 4. The stable storage layer provides the required stable storage mechanism described in section 2.3. It also supports a uniform addressing mechanism over the stable storage that corresponds to the virtual addressing abstraction described in section 2.1.3. In practice the virtual addressing is supported by lower level addressing mechanisms that give access to the non-volatile storage, the main memory and any other physical storage devices provided by the underlying hardware.

The interface to the stable storage has been designed to provide a contiguous range of virtual addresses that is always in a self consistent state. This is achieved by implementing a checkpointing mechanism that preserves the current state of the store on non volatile storage.

At any point in time the non volatile storage contains a self consistent version of the store. The act of performing a checkpoint replaces the previous recorded state in a single atomic action. When a failure occurs the store is automatically restored to the state recorded by the most recent checkpoint. This simple checkpointing mechanism is sufficient to simulate a stable store.

Although the semantics of the required checkpointing mechanism are simple, the actual implementation may be quite sophisticated. To accommodate as much flexibility as possible the interface includes a set of procedures that allow the use of the virtual address space to be dynamically configured. For example, the implementation of the heap layer may use some temporary data structures that are reconstructed each time the system is restarted. In this case, changes to the storage containing these data structures need not be recorded between checkpoints and the data itself need not be recorded by a checkpoint. In contrast, any changes to user data must be recorded between checkpoints to support the reconstruction of the previous consistent state and the new values of the data must be recorded by a checkpoint.

The range of storage uses that are supported include:

- Read-only This is the default state for all user data.
- Save-only This describes an area of store that must be saved at the next checkpoint but it does not form part of the previous checkpoint.
- Shadow All changes to the specified area of storage must be recorded. It contains data that is part of the previous checkpoint and must be part of the next checkpoint. This requires the allocation of non volatile storage to record any changes.
- Scratch The specified area of storage is for use by temporary data. The data is not part of the previous checkpoint and need not be protected from store failures.
- Reserve The specified area of storage may be required following the next checkpoint operation. It must be allocated non volatile storage but the storage may be used for other purposes prior to the next checkpoint.
- Not-required The area of storage is no longer required to contain data. The non volatile storage allocated to the area may be reallocated for other purposes.

Given this detailed information on the desired use of the virtual address space the layer implementation may be able to optimise its checkpointing and storage allocation strategies. Thus it may be possible to use the available physical resources to their full effect.

4.4. Napier88

To conclude we shall now briefly describe the persistent programming language Napier88 and how it has been implemented using the layered architecture.

Napier88 is a persistent programming language with a sophisticated type system that permits the recursive definition of data structures including abstract data types and polymorphic first

class procedures. As far as possible the Napier88 compilation system performs static type checking. That is, the compiler will determine whether or not an attempted operation is type correct. However, there are certain situations where this is not possible.

Firstly, dynamic checks are generated by the compiler for vector indexing operations and field updates to dynamically created data structures. The first check is to ensure that a vector index is legal and the second check ensures that constant locations are not updated. Neither of these checks may be statically determined from the type system.

The second situation that cannot be statically checked is the use of a value from an infinite union. Napier88 provides a type **any** that is the infinite union of all data types. A value obtained from a variable of type **any** must be projected onto its actual type before it can be used. The projection must be performed dynamically since, in general, it is not possible to statically determine the actual type of the value.

To aid the separate preparation of programs and data the type checking is based on structural equivalence. This allows the Napier88 system to perform dynamic type checking without the use of a centralised type dictionary. However, if available, a centralised dictionary can be used to optimise the dynamic checking.

Napier88 supports first class procedures via a block retention mechanism. The block retention mechanism implements a program stack with a separate object for each activation record. A garbage collector is relied on to automatically determine which activation records are not part of a procedure closure and can be discarded. Consequently, the block retention mechanism can be directly supported by a persistent object store without the need to provide large extensible objects to model a stack.

Polymorphism and abstract data types are supported by a combination of compilation abstraction and a set of adhoc primitive operations. The adhoc primitives use an integer key to identify the size of stack elements to manipulate and the rules for performing equality. The integer keys are made available to polymorphic code as part of the static environment provided by the block retention.

The Napier88 compilation system maps programs onto an abstract machine [7]. The abstract machine is based on block retention and is responsible for implementing those primitives necessary to support the polymorphism and abstract data types. In turn, the abstract machine views the persistent object store as a single heap of persistent objects that is assumed to be a stable store of unbounded size. Since the abstract machine does not allow direct access to the persistent store it ensures that the compilation system is unaware of the implementation of the object storage thus separating the use of an object from the way it is stored.

4.5. The Napier88 Implementation

The implementation of Napier88 on the layered architecture is a simple matter of interfacing the abstract machine to the persistent object store. The resulting architectural layers are shown in Figure 5.

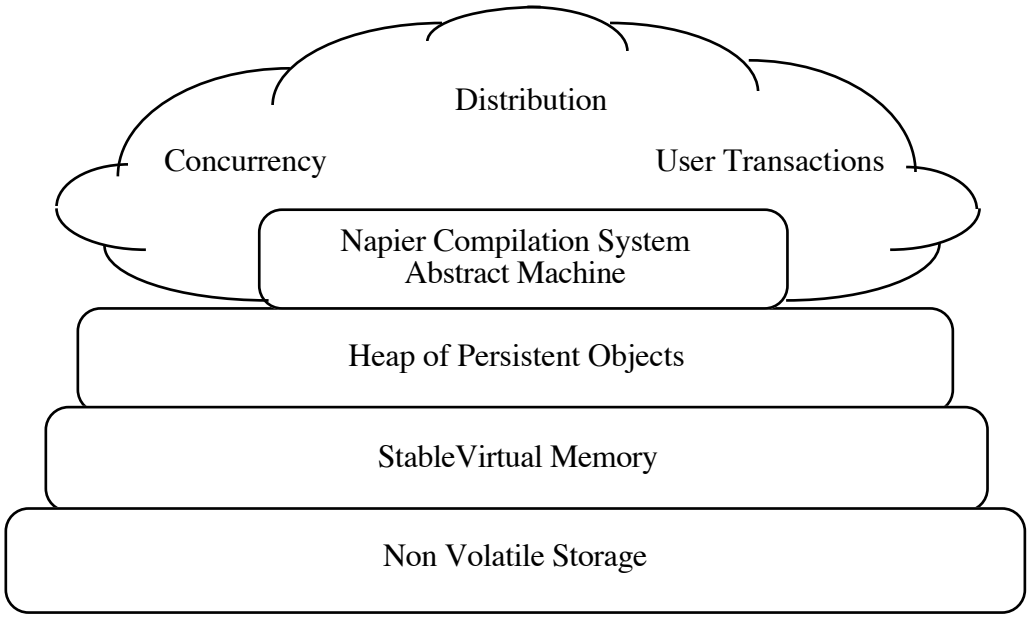

Figure 5: The Architectural Layers Used to Implement Napier88

The Napier88 compilation system ensures that all operations attempted by a program are type correct. The operations that perform the dynamic checking are also type correct and are implemented by language level operations. Thus, the Napier88 compilation system performs the task of a high level protection mechanism.

The abstract machine to which Napier88 is compiled operates against the heap layer of the architecture and adheres to the conventions specified by the heap interface. For efficiency reasons it maintains some data and object addresses in special purpose registers. However, in keeping with the interface specification it copies all the cached data back to heap objects prior to requesting garbage collection or checkpoint operations.

Since the compilation system provides a high level protection mechanism the heap layer of the architecture does not attempt to enforce any form of protection. Thus, it is able to perform the task of organising the stable storage without the support of special purpose hardware. It is also possible to optimise the addressing of objects since the attempted operations may be assumed to be type correct.

There are currently two functionally equivalent implementations of the stable storage layer that support the heap layer. One implementation performs its own address translation and input/ output buffering. The resulting performance is poor but is acceptable if the abstract machine maintains an object cache in main memory. An alternative implementation is available on Sun workstations using memory mapped files. This implementation performs all addressing using the Sun memory management hardware to provide efficient access to the stable storage.

The efficiency of the second implementation has been greatly enhanced through the effective use of the dynamic configuration procedures provided by the stable storage interface described in section 4.3. In future implementations increased control over the paging algorithms will further improve the system performance. Thus, it is possible to construct an efficient implementation of a persistent object store on conventional hardware.

5. CONCLUSIONS

We have briefly described the architectural mechanisms required to support a type secure persistent object store and shown how they may be modelled as separate architectural layers. A layered architecture has also been described that provides the architectural mechanisms as separate architectural layers that must conform to a specified interface. The benefits of the layered architecture include the ability to easily construct experimental systems based on new implementation techniques for one or more of the architectural layers and the ability to construct efficient implementations on conventional hardware where a high level protection mechanism may be employed.

Examples of the layered architecture include the Napier88 system described above and the Staple functional programming system. Implementations of the stable storage and heap layers are available on Sun workstations and the Apple Macintosh for experimental use.

ACKNOWLEDGEMENTS

This work was undertaken at St.Andrews University during the study leave period of Prof. J. Rosenberg of the University of Newcastle, New South Wales. The work was supported by SERC Grant GR/F 28571, SERC Post-doctoral Fellowship B/ITF/199, ESPRIT II Basic Research Action 3070 - FIDE and a grant from the DTI.

REFERENCES

1. 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.

2. 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.

3. 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.

4. 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.

5. Beloff B., McIntyre D. & Drummond B. "Rekursiv Hardware." Linn Smart Computing Ltd, 1988.

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

7. 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.

8. Brown A.L. (Ph.D. Thesis) "Persistent Object Stores." Universities of Glasgow and St.Andrews PPRR-71, Scotland, 1989.

9. Buckle J.K. *The ICL 2900 Series*, Macmillan Computer Science Series, Macmillan, 1978.

10. 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.

11. Cohen J. "Garbage Collection of Linked Data Structures." *ACM Computing Surveys*, vol. 13, no. 3, 1981, pp341-367.

12. Daley R.C. & Dennis B.D. "Virtual Memory, Processes and Sharing in MULTICS." *Comm. ACM* vol. 11, no. 5, 1968, pp306-312.

13. Goldberg A. & Robson D. Smalltalk-80: The language and its Implementation. Addison Wesley, 1983.

14. 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.

15. Kernighan B.W. & Ritchie D.M. *The C programming language*. Prentice-Hall, 1978.

16. Knowlton K. "A Fast Storage Allocator." *Comm. ACM*, vol 8, 1965, pp623-625.

17. 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.

18. Lorie A.L. "Physical Integrity in a Large Segmented Database." *ACM Transactions on Database Systems*, vol. 2, no. 1, 1977, pp91-104.

19. McCarthy J. "Recursive Functions of Symbolic Expressions and their Computation by Machine." *Comm. ACM*, vol. 3, no. 4, 1960, pp184-195.

20. McNally D.J. "Code Generating Functional Language Modules for a Persistent Object Store." Staple Project Research Report, Staple/StA/89/2, University of St.Andrews, Scotland, 1989.

21. Morrison R. "S-algol: A Simple Algol." *BCS Computer Bulletin Series II*, no. 31, March 1982, pp17,20.

22. Morrison R., Brown A.L., Connor R. & Dearle A. "The Napier88 Reference Manual." Universities of Glasgow and St.Andrews PPRR-77, Scotland, 1989.

23. Organick E.I., *Computer System Organisation: The B5700/B6700 Series*, Academic Press, New York, 1973.

24. Perry N. "Hope+." Imperial College Internal Report IC/FPR/LANG/2.5.1/7, 1987.

25. "The PS-algol Reference Manual fifth edition." Universities of Glasgow and St.Andrews PPRR-12, Scotland, 1988.

26. 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.

27. Rosenberg J., Henskens F., Brown A.L., Morrison R. & Munro D.S. "Stability in a Persistent Store Based on a Large Virtual Memory." *International Workshop on Computer Architectures to Support Security and Persistence*, Universität Bremen, West Germany, May 1990.

28. Ross G.D.M. (Ph.D. Thesis) "Virtual Files: A Framework for Experimental Design." University of Edinburgh, 1983.

29. 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.

30. 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.

31. Wirth N. "The programming language Pascal." *Acta Informatica*, vol. 1, no. 1, 1973, pp35-63.