A Generic Persistent Object Store

A.L. Brown & R.Morrison

Department of Mathematical and Computational Sciences,
University of St.Andrews,
North Haugh,
St.Andrews.
KY16 9SS

Abstract

Persistent programming systems have demonstrated the benefits of abstracting over the implementation details related to the storage and manipulation of long term data. In particular, programming systems in which persistence is provided as an orthogonal property of all data types have been used to produce cost-effective implementations of applications such as integrated programming support environments. In this paper, a generic architecture which supports persistent programming is described. In particular the provision of an object store is highlighted, outlining its design aims and resultant structure. The store is designed as a modularised tool set with a high degree of reuseability as a main design goal. The application of the architecture in a wide variety of programming paradigms is reported and it is argued that the architecture is a cost-effective technique for integrating persistence and the programming paradigm best suited to a particular application system.

1. INTRODUCTION

Persistent programming systems are aimed at reducing the cost, both intellectual and mechanical, of producing data intensive application systems. These application systems arise in areas such as software engineering, CASE tools, CAD/CAM, office automation, document preparation and object oriented systems. The common element among these applications is that the objects that are being manipulated are in some sense complex and, by the nature of the applications, are unsuited to processing by more conventional database techniques.

An essential feature of a persistent programming system is a store capable of supporting complex objects. A number of proposals and implementations of persistent stores have been made [1, 2, 3, 4, 5, 6, 7, 8]. Each store has its own set of design goals since each has been used with a particular view of the persistence abstraction.

The aim of this work is to provide a suitable platform upon which programming language implementors may build a persistent version of their particular system. The hypothesis is that persistence is orthogonal to the programming paradigm and should be available as an abstraction for all systems [9]. Towards this view a platform is provided in a manner that can be tailored to the particular system. This is achieved by providing a set of reuseable components within a generic persistent architecture. Subgoals in providing the platform address issues such as extensibility, performance, stability and experiments with concurrency, transactions and distribution.

In this paper a generic persistent architecture which provides a persistent object store is described, outlining its design aims and resultant structure. Since the generic architecture relies on mapping a programming language onto objects, it may be applied to any programming system which may be implemented using such a mapping. To date, the generic architecture has been employed to provide persistence to a wide variety of programming paradigms including imperative programming with first class procedures [10], functional programming [11], typefull programming [12] and object-oriented database programming [13]. Consequently, it is argued that the generic architecture provides a cost-effective technique for integrating persistence and the programming paradigm best suited to a particular application system.

2. THE ARCHITECTURAL DESIGN

The initial motivation for the design of the generic architecture was to support the persistent programming language Napier88 [10]. Previous experience in building persistent object stores led to a concentration on two new requirements for the overall architecture, high performance and experimentation with implementation techniques. It was envisaged that the Napier88 system would be used as the basis for future experiments in concurrency, distribution and transactions. Thus, the design had to include support for these future experiments without attempting to prejudge the particular conceptual models.

In order to reduce the cost of experimentation a clear design aim is to avoid vertical structure within the architecture. That is, as far as practicable, changes to a particular architectural component should not result in secondary changes to every other architectural component. For example, if an object format is known to each component in a system such as garbage collectors, transaction mechanisms, abstract machines and compilers, then a change to the object format requires consistent changes to each component of the architecture. This form of vertical structure imposes a potentially high cost on experimentation.

The second major design aim was to avoid the need for highly structured interfaces between architectural components. A major performance overhead in many systems is the complexity of moving data around the storage hierarchy. For example, if data is held in a database system it must initially be accessed using the database's own interface procedures. Subsequently additional conversion procedures may need to be employed to convert the data into the desired form. The approach adopted by the generic architecture is to allow direct access to permanent data but to recreate the logical structure by appropriate levels of abstraction. Ideally the levels of abstraction are realised by convention thereby reducing the potential performance overheads.

It soon became clear that the architecture for the Napier system was suitable for supporting other languages. By making the architecture generic in the form of a modularised plug in tool set and providing appropriate components to plug in, then any language architecture may be supported. We should however qualify this by saying that not all system components for all languages have been written. A small and encouraging measure of success is the diversity of language paradigms that have been supported, mostly on the Napier base.

In summary, the generic architecture was designed to support:

- a persistent programming system,
- cost-effective experiments in concurrency, transactions and distribution, and
- direct access to permanent storage as an aid to high performance.

2.1 A Persistent Object Store

Before describing the generic architecture the persistent object store design is outlined. The key elements in providing a persistent object store are:

- the identification of persistent objects,
- the properties of objects and their interconnections,
- the scale of the object store, and
- the provision of stability.

Each of these will be taken in turn.

2.1.1 The Identification of Persistent Objects

A key design decision in all persistent programming systems is how long term data, that is persistent data, should be identified. A variety of different techniques are employed in existing systems and include:

- core-dumping,
- explicit marking, and
- reachability from a root of persistence.

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 and is used by Smalltalk [14] and some Lisp systems [15].

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 [16].

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 [17].

In practice many systems employ a mixture of these techniques. For example, in the Argus system [18] explicit marking is 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.

An alternative to employing a combination of techniques is to employ a single technique and simulate the other two from it. For example, if reachability from a root of persistence is adopted then core-dumping may be simulated by ensuring that all usable data is always reachable from the root of persistence. Similarly, explicit marking may be simulated by ensuring that a reference to explicitly marked data is always held within data which is itself reachable from the root of persistence. Reachability will then automatically treat the marked data as reachable.

Since identifying persistent data by reachability is the most flexible it was chosen for adoption within the generic architecture.

2.1.2 Persistent Objects and Their Relationships

The applications that most benefit from orthogonal persistence have the property that they manipulate complex objects. Complex objects are objects that are not flat, like first order relations, and may explicitly reference other objects. The store must provide a notion of object appropriate to the implementation of these applications. The objects themselves may form a potentially complex interconnected graph. It is important that the topology of the graph is preserved between programs since, for example, an index constructed in one program may be preserved and reused in another. Indeed such reuse of structure is a major efficiency gain of persistence.

To accommodate a wide variety of programming paradigms it is important to keep both the objects and their interrelationships simple at the store level. Thus objects consist of a contiguous area of storage which, by convention, is subdivided between pointers and values. Furthermore, some housekeeping information is also recorded in order to facilitate garbage collection and the identification of persistent data by reachability. The interrelationships between objects are maintained by pointers which denote the identity of an object.

This simple approach is in direct contrast to other stores which may have type information, method invocation information, hierarchy information, etc., preserved with the object and more complex relationships stored with the pointers. This choice is deliberate since models of type, inheritance, method invocation and complex relationships vary widely from system to system. Building these ideas into the store would have been in direct opposition to the goal of experimentation and it had been precisely this experience that had led to the abandoning of vertical integration.

2.1.3 The Scale of the Object Store

In many programming languages it is possible to dynamically create values within a conceptually infinite store. The generic architecture must therefore have the ability to support the simulation of such a conceptually unbounded store.

Two techniques commonly used to simulate an unbounded store are a stack and a heap. With a stack, stack frames are automatically created and reclaimed in step with procedures calls and returns. In contrast, objects are created within a heap when required and, in the presence of garbage collection, are automatically reclaimed only when it is clear that they are no longer usable, that is they are no longer reachable from one or more roots within the heap.

Of these two techniques the heap is more general in that stack frames may be easily modelled as objects within a heap. Also, in the case of first class procedures, the stack frames may be retained for as long as they are required. A third advantage of a heap is that the task of identifying garbage for collection is similar to the task of identifying persistent data in that both tasks are based on reachability. For these reasons the generic architecture presents the persistent store as a heap of objects with garbage collection.

As outlined above, it is important that the persistent store preserves the topology of the relationships between objects and the identity maintained by the pointers. A convenient way of achieving this is to ensure that all access to the persistent store is made via a single consistent view. Consequently, the persistent store provided by the generic architecture may only be viewed as a heap of objects.

One final advantage of providing a uniform view of the persistent store is that store exhaustion can only occur by one means. It is important that when the persistent object store is full then the user knows about this. However, the system must only indicate that the store is full when it really is and not because one component has exhausted its own store. Such a situation often arises in mixed stack and heap architectures.

2.1.4 Stability

All application systems that handle inherently valuable data must provide facilities to safeguard the data from accidental loss or inconsistent update. Thus, support must be provided both to recover from unexpected failures of a system and to allow an appropriate form of transactional update within a system.

Unexpected failures may be accommodated by always ensuring that the store may be restored to a recent self-consistent state. The known techniques for achieving this are based on data copying ranging from regular dumping onto non-volatile storage to maintaining multiple on-line copies [19, 20, 21]. The cost of these techniques is proportional to the frequency with which self-consistent states are recorded and the desired speed of reconstruction following a failure. A store which incorporates one or more of these techniques may be said to exhibit stability.

Transactional updates within a system are usually assumed to perform two complementary tasks [22]. Firstly, they are required to ensure that either all the desired updates are performed as a single atomic action or none of them are. Thus, a transaction is an important tool in ensuring that all updates to a system are logically consistent. Transactions which perform this task may be described as user transactions. Secondly, on completion of a transaction it is often assumed that the updates have been successfully saved to non-volatile storage and will survive a system crash.

The support required for the second task is similar to that required to recover from unexpected failure. As a result, it may be argued that it is sufficient to provide the ability to recover from unexpected failures together with a facility to ensure that the current self-consistent state of the system is recorded on non-volatile storage. In this way, a user transaction may ensure that the log of operations it wishes to perform has been safely recorded on non-volatile storage before applying them.

This approach is taken by the generic architecture with the result that the persistent object store is presented as a heap of objects that is resilient to unexpected failure and is able to support the implementation of user transactions.

3. THE GENERIC ARCHITECTURE

The generic persistent architecture is now presented. It is designed as a modularised tool set with a high degree of reuseability. Each module constitutes a layer in the architecture. The layered architecture provides a uniform addressing mechanism, a storage management mechanism, a stability mechanism and a protection mechanism within a set of distinct architectural layers.

3.1 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. 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 user 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 1.

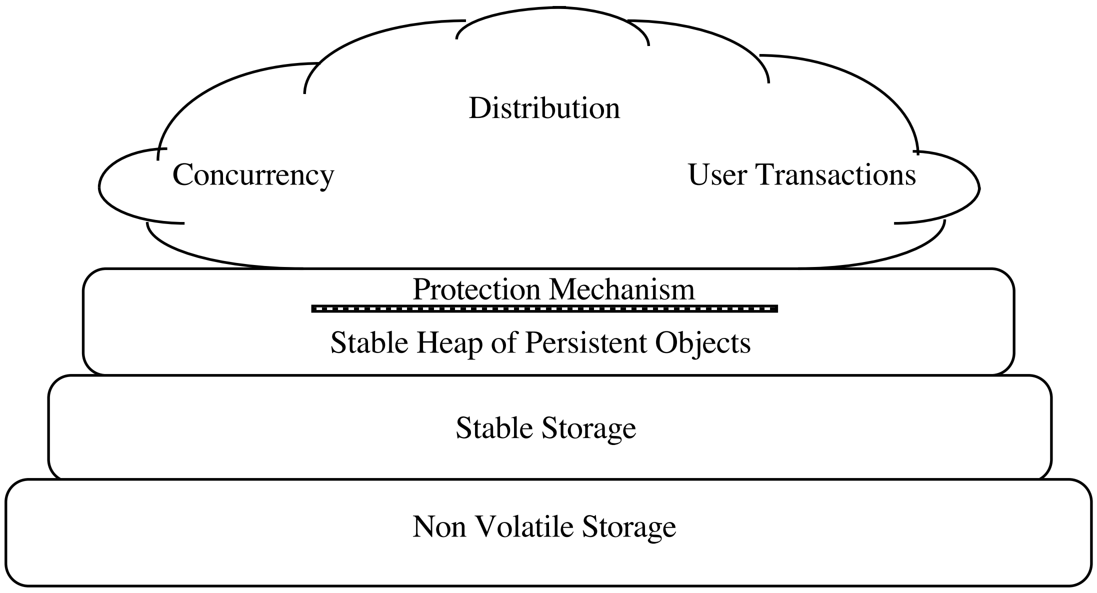

Figure 1: The basic architectural layers.

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.

3.2 The Stable Heap

The layer of the persistent object store which is visible to the programming language level, is the stable heap of persistent objects shown in Figure 1. 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, this is described more fully below. 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.

At the lower levels of the persistent store, stability is simulated by a simple checkpointing mechanism. This mechanism is visible 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 functions 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.

3.2.1 The Stable Heap Interface

The stable heap interface provides a set of interface functions which include:

- create_object, set_lock,
- read_word, write_word, pointer_to_address,
- garbage_collect, stabilise, first_object,
- can_modify,
- open, and configuration.

3.2.1.1 Create Object

The create_object interface function provides a mechanism for creating new objects and obtaining new pointer values. All objects in the stable heap are an integral number words in length. The first word, word 0, contains the number of pointer fields and the second word, word 1, contains the total size of the object in words. Following the two header words are all the pointer fields, one per word. The interpretation of the remainder of an object is the responsibility of the user program. The object format is illustrated in Figure 2.

In order to support the implementation of concurrency an object locking protocol is provided. Every object is prefixed by a lock word which is manipulated using the interface function set_lock. A lock is recorded by negating the lock word using atomic test and set operations and a lock is cleared by setting the lock word to a positive value.

Lock-Word	Number of Pointers (m)	TotalSize in Words (n)	(m) Pointers	(n-m-2) Non-Pointers
Word -1	Word 0	Word 1	Words 2..m-2	Words m-1..n-1

Figure 2: The object format.

3.2.1.2 Reading and Writing Objects

The read_word and write_word functions are provided to allow the higher level architecture to address data purely in terms of pointers to objects and indexing of objects without requiring any explicit knowledge of how the pointers may be mapped to physical addresses. These two functions are both parameterised by a pointer and a word index within the referenced object.

To avoid the need for explicit function calls the interface also provides a mapping function from pointers to physical addresses. Using the result of this mapping a user program may directly access the contents of an object without incurring the overhead of function calling. However, the result of the mapping only remains valid until the next garbage collection or system restart. Thereafter the mapping must be re-established.

3.2.1.3 Garbage Collection and Stability

As mentioned above, the interface provides an explicit interface function, stabilise, to invoke a checkpoint of the persistent store. One advantage of this is that the higher level architecture is able to cache data outwith the stable heap and need only replace it prior to the call of stabilise. For similar reasons the stable heap's garbage collector must be explicitly invoked. Thus, prior to garbage collection the higher level architecture can replace any cached data thereby providing an up to date consistent view of the stable heap to the garbage collector. Furthermore, the higher level architecture can also take steps to ensure that any mappings from pointers to physical addresses are re-established after the garbage collection.

The garbage collector identifies garbage by following pointer fields starting from a root object. Any objects that are not reachable from the root object are garbage and the storage allocated to them will be reused. Thus, all usable data must be reachable from the stable heap's root object if it is to survive garbage collection. To support this, the stable heap provides an interface function, first_object, which returns a pointer to the root object. The first pointer field in the root object is available to a user program for referencing user data. In practise this pointer field points at a root object for the user program's own data.

3.2.1.4 Modifying Objects

In order to implement the checkpointing mechanism invoked by stabilise, the persistent store must maintain an area of storage in which it can record those changes that occur between checkpoints. In many cases this may be insufficient to allow the entire stable heap to modified. Thus, at some unspecified point a program may have to call stabilise before it can continue. To allow a program to detect the point at which stabilise must be called, an interface function can_modify is provided. Prior to modifying an object can_modify may be called to establish that there is indeed sufficient storage available to record any changes to that object. Failure to employ can_modify may result in a user program unexpectedly failing.

3.2.1.5 Initialisation and Configuration

On each use of the stable heap a user program must open the persistent store and initialise the interface functions. To support this an interface function, open, is supplied which is parameterised by the name of the persistent store, and some user defined functions. The user defined functions include an error handler that is called with a string parameter if a run-time error occurs in the stable heap. A handler function is also supplied that is invoked when an unexpected call to stabilise must be made. Such unexpected calls may be required if objects are directly accessed without using can_modify or if a garbage collection involves more modifications to the persistent store than can be accommodated by the checkpointing mechanism.

In the interests of efficiency some user programs may wish to make certain assumptions regarding the implementation of the persistent store. For example, a program may wish to directly map pointers to physical addresses without using the interface function. Other programs may wish to store tagged values in pointer fields that are not pointers: the garbage collector ignores values in pointer fields which cannot be pointers. In both cases a mechanism is required to accommodate this while preserving the integrity of the stable heap.

This is achieved by providing an interface function, configuration, which details any assumptions a user program may make about the actual implementation in use. The information reported by configuration includes:

- the range of values for valid pointers,
- how a pointer is mapped onto a physical address,
- what effect garbage collection may have on pointer values, and
- whether or not a mapping from pointers to physical addresses is implemented.

Given this configuration information a user program can ensure that it does not make any invalid assumptions regarding the particular implementation of the persistent store it is using. If a program depends on certain assumptions it should not attempt to use the store unless they are valid.

3.2.2 Protection

The heap layer presents the higher level architecture with a view of 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 computer [23] 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, via the interface function configuration, to ensure that an instance of the architecture is composed from compatible layer implementations.

3.3 The Stable Store

The heap layer is directly supported by a single contiguous stable store, see Figure 1. The stable storage layer provides the required stable storage mechanism described in section 2.1.4. It also supports a uniform addressing mechanism over the stable storage that provides a virtual addressing abstraction. 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.

3.3.1 The Stable Store Interface

The stable storage interface provides a set of interface functions which include:

- first_address, last_address, real_address
- read_word, write_word, set_lock,
- read_only, save_only, shadow, scratch, reserve, not_required,
- stabilise, and open.

3.3.1.1 Addressing Stable Storage

The stable storage layer reports the range of addresses which it maintains in a self consistent state via the interface functions first_address and last_address. The results of these functions are the address of the first and last words of stable storage that may be addressed by the higher level architecture. This information dictates the location of the stable heap's data structures.

Access to the stable storage may be achieved via the interface functions read_word and write_word which support access a word at a time. A similar function set_lock is available which implements write_word via atomic test and set instructions.

The addresses used by the stable storage may not correspond to the physical addresses available to a user program. So, to enable direct access to the stable storage an interface function, real_address, is provided to map stable storage addresses onto physical addresses. Given this function the stable storage may be directly accessed without the overhead of calling the read_word and write_word interface functions. Furthermore, it is then possible to implement the stable heap's interface function pointer_to_address.

3.3.1.2 Dynamic Storage Configuration

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.

An interface function is provided for each of these possible storage uses and is parameterised by the addresses of the first and last words of stable storage that are affected. 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.

3.3.1.3 Initialisation

On each use of the persistent store the initialisation of the stable heap's interface will also cause the stable store's interface to be initialised. This is supported by an interface function, open, which is parameterised by the name of the persistent store and some stable heap defined functions. These stable heap defined functions include an error handler that is called with a string parameter if a runtime error occurs in the stable store. A call of the error handler would in turn cause a call to the user defined error handler initially supplied to the stable heap interface. A write-fault handler function is also provided which is invoked if an update to a word of stable storage has not been authorised by a previous call to one of the interface functions save_only, scratch or shadow.

Following a call to the write fault handler a user program cannot resume until the appropriate dynamic configuration function has been called. This would be performed automatically by the handler supplied by the stable heap. However, a call to one of these functions may require the allocation of non-volatile storage to record the pending update. If insufficient non volatile storage is available then the configuration function will fail and the handler must then invoke a checkpoint via the interface function stabilise.

As described above, the persistent store must not invoke a checkpoint automatically since the user program may have cached data outwith the store. Therefore, the write fault handler must then invoke the user defined function supplied to the stable heap as the mechanism for requesting unexpected calls to stabilise. If a checkpoint is subsequently invoked then the dynamic configuration can be retried and the user program finally resumed.

4. CONCLUSIONS

The generic architecture described above has been aimed at providing a suitable platform upon which programming language implementors may build a persistent version of their particular system. Towards this, the architecture provides a set of reuseable components that can be tailored to a particular system. The subgoals in providing the platform address issues such as extensibility, performance and cost-effective experimentation.

The use of the generic persistent architecture relies on mapping a programming language onto objects. Thus, it may be employed with any programming system which may be implemented using such a mapping. To date, implementations of the generic architecture have been completed for the Napier88 system [24], the functional programming language Staple [25], the object-oriented database programming language Galileo [26] and a typefull programming language Quest [27]. Each of these implementations has demonstrated the cost-effectiveness of using the generic architecture to integrate persistence with a particular programming paradigm.

The issue of high performance is addressed in the provision of configuration interfaces that allow a user program to directly access permanent storage within the persistent store. This permits efficient access to data without the overhead of function calls via an interface but, the logical structure of the interfaces is retained by appropriate levels of abstraction.

In order to reduce the cost of experimentation each architectural component within the generic architecture must conform to a specified interface and any changes to an individual interface need only affect the component that directly accesses the changed interface. The advantage of this horizontal or layered approach is that individual components may be independently reimplemented without incurring a knock on effect. The advantages of the layering are even preserved where an interface must be changed since the propagation of changes is limited to the component that uses the changed interface.

The design of the generic persistent architecture is the focus of continuing research into the best interfaces to meet the requirements of the different persistent application systems. In particular the support required for a fully distributed persistent programming system is under active consideration based on the experience with the existing implementations.

5. ACKNOWLEDGEMENTS

The work was supported by SERC Grants GR/F 28571, GR/G 17578 and GR/G 33011, SERC Post-doctoral Fellowship B/ITF/199, ESPRIT II Basic Research Action 3070 - FIDE and a grant from the DTI Awareness Initiative in Object Oriented Programming.

6. REFERENCES

1. 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.

2. Cockshott W.P., Atkinson M.P., Chisholm K.J., Bailey P.J., & Morrison R. POMS: A Persistent Object Management System. Software Practice and Experience, vol. 14, no. 1, 1984, pp49-71.

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

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

5. Moss J.E.B., & Sinofsky S. Managing Persistent Data With Mneme: Designing a Reliable Shared Object Interface. In Advances in Object-Oriented Database Systems, Lecture Notes in Computer Science, Springer-Verlag, vol. 334, 1988, pp298-316.

6. Skarra A., Zdonik S.B. & Reiss S.P. An Object Server For An Object-Oriented Database System. Proc. International Workshop on Object-Oriented Database Systems, Pacific Grove, California, 1987, pp196-204.

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

8. 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.

9. 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.

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

11. Davie A.J.T. & McNally D.J. Statically Typed Applicative Persistent Language Environments (STAPLE) User's Manual. Computational Science Research Report, CS/90/14, University of St.Andrews, Scotland, 1990.

12. Cardelli, L. Typeful Programming. DEC SRC Report, May 1989.

13. 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.

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

15. 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.

16. Cardelli, L. Amber. Tech. Report AT&T. Bell Labs. Murray Hill, U.S.A., 1985.

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

18. 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.

19. 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.

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

21. 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.

22. Atkinson M.P. & Morrsion R. Integrated Persistent Programming Systems. Proc. Hawaii International Conference on System Sciences, 1986.

23. Harland D.M. Rekursiv Object-Oriented Computer Architecture, Ellis Horwood, 1988.

24. Brown, A.L. & Rosenberg J. Persistent Object Stores: An Implementation Technique. Proc. Fourth International Workshop on Persistent Object Systems: Their Design Implementation and Use. Martha's Vineyard, Massachusetts, USA, September 1990.

25. McNally D.J., Joosten S. & Davie A.J.T. Persistent Functional Programming in the Large. Proc. Fourth International Workshop on Persistent Object Systems: Their Design Implementation and Use. Martha's Vineyard, Massachusetts, USA, September 1990.

26. Mainetto G. & Brown A.L. Integrating Galileo with a Persistent Object Store. (In preparation)

27. Matthes F., Brown A.L. & Schmidt J. Integrating Quest with a Persistent Object Store. (In preparation)