This paper should be referenced as:

Morrison, R. & Atkinson, M.P. “Persistent Languages and Architectures”. In **Security and Persistence**, Rosenberg, J. & Keedy, J.L. (ed), Springer-Verlag (1990) pp 9-28.

Persistent Languages and Architectures

R Morrison
University of St Andrews

M P Atkinson
University of Glasgow

ABSTRACT

Persistent programming is concerned with creating and manipulating data in a manner that is independent of its lifetime. The persistence abstraction yields a number of advantages in terms of orthogonal design and programmer productivity. One major advantage is that the abstraction integrates the database view of information with the programming language view. For this reason **persistent programming languages** are sometimes called *database programming languages*.

A number of design principles have been devised for persistent systems. Following these rules, languages may be designed that provide persistence as a basic abstraction. This in turns begs the question of how these languages should be implemented and what architectural support is required for persistence.

Here we will review the concepts of persistence and re-examine the design issues that appear in persistent languages and architectures.

1. INTRODUCTION

Our aim is to support the activity of applications construction. Currently the underlying technology relies on a number of disparate mechanisms and philosophical assumptions for support and efficient implementation. Among these are:

- a plethora of disparate mechanisms is used in constructing applications [6] - command languages, editors, file systems, compilers, interpreters, linkage editors, binders, debuggers, DBMS - DDLs and DMLs, query languages, graphics languages, transaction managers, concurrency models, machine types etc.
- a reliance on the computing system as a data store or a data processor and separating the models.
- a late 60s and early 70s models of computation based on small, volatile, fast main stores and large, non-volatile, slow rotating technologies.

The incoherence and complexity arising from utilising many different and diverse application and system building mechanisms increases the cost both intellectually and mechanically of building even the simplest of systems. The complexity distracts the application builders from the task in hand forcing them to concentrate on mastering the programming systems rather than the application being developed. Perversely the plethora of disparate mechanisms is also costly in machine terms in that the code for interfacing them causes overheads in both space and time to execute.

Most application support systems, such as the Ada APSE, rely on a model of computation that separates program and data. Programs are stored in program libraries and data is stored in the filing system or a database management system. More modern modelling techniques such as object-orientation rely on the fact that program and data are intimately bound. This mismatch between the language and the modelling technique adds complexity to the programming system.

Finally, the model of computation based on small, volatile, fast main stores and large, non-volatile, slow rotating stores, which was developed in the late 1960's has survived and still influences the manner in which applications are constructed.

The advent of new hardware technologies with different time, space and cost trade-offs together with the development of new application areas, such as office automation, CAD/CAM, CASE etc. has led to proposals for different models of computation both linguistically and architecturally. Persistent systems are one such proposal. The persistence abstraction allows some of the complexity to be removed thereby reducing the cost of designing, implementing and maintaining applications using long term data.

In the following sections we will review some of the tradition of persistence and look at the challenges for the future.

2. PERSISTENCE DEFINITION

We defined the persistence of data is the length of time for which the data exists and is usable [6]. A spectrum of persistence exists and is categorised by

- transient results in expression evaluation,
- local variables in procedure activations,
- own variables, global variables and heap items whose extent is different from their scope,
- data that exists between executions of a program,
- data that exists between various versions of a program, and
- data that outlives the program.

The first three categories are usually provided by a programming language whereas the last three are provided by a filing system or a database management system.

A persistent programming system accommodates all categories of longevity in data. We aspire to systems where the use of the data is independent of its persistence.

2.1. Principles of Orthogonal Persistence

We have also identified the following principles which define the persistent abstraction. They are:

- ***The Principle of Persistence Independence***

  The persistence of data is independent of how the program manipulates the data. That is, the user does not have to, indeed cannot, program to control the movement of data between long term and short term store. This is performed automatically by the system.

- ***The Principle of Data Type Orthogonality***

  All data objects should be allowed the full range of persistence irrespective of their type. That is, there are no special cases where objects of a specific type are not allowed to be persistent.

- ***The Principle of Persistence Identification***

  The choice of how to identify and provide persistent objects is orthogonal to the universe of discourse of the system. That is, the mechanism for identifying persistent objects is not related to the type system.

The application of the three principles yields ***Orthogonal Persistence.***

2.2. Loss of Orthogonality

Loss of orthogonality of persistence occurs by disregarding any of the three principles. Most serious is persistence independence since if it is broken it is hard to see how the persistence in the system is orthogonal in any way.

Some programming language designers have taken a very pragmatic approach to persistence and disregarded data type orthogonality for specific types. Pascal/R [41] and DBPL [29] are persistent programming languages where only first order relations are persistent and then only relations that do not contain pointer types. This models the relational world well but causes difficulty when other modelling techniques are used.

The identification of persistent objects is commonly performed by the system automatically computing the transitive closure of objects from some persistent root [4,5,13,14,18]. Where this is not the case languages often associate persistence with type. This breaks the third principle and as a side effect the second one as well. It also gives rise to dangling reference problems, or at least invalidated references, for persistent objects that point to non-persistent objects. The E programming language [37], nearly all the persistent extensions to C^{++} [43] and the PGraphite language [46] use this technique. Figures 1 and 2 illustrates the problem.

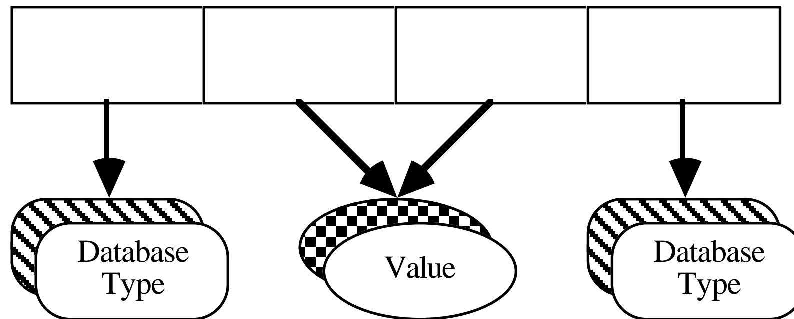

Figure 1: Persistent objects before being sent to the persistent store.

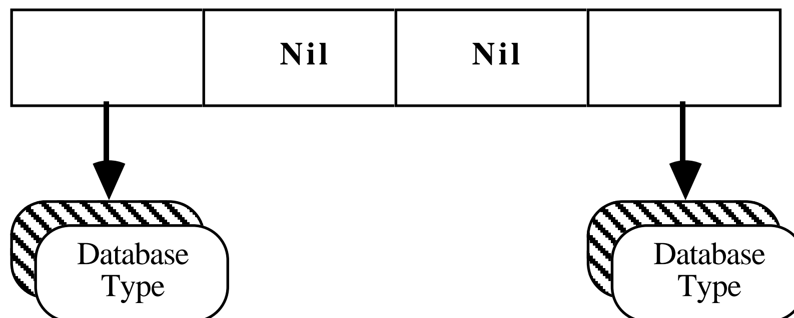

Figure 2: The same objects as in Figure 1 in the persistent store.

Figure 1 shows an object which points to two other objects of database types and a non-database value before being preserved in the persistent store. Figure 2 shows that in the persistent store only the database types are preserved and the non-database value is lost, being replaced by a nil value if such is available or a dangling reference. Here the persistent store model of data is not consistent with the main store model, a position we wish to avoid since this adds complexity to the system for the programmer to master.

2.3. Savings with Persistence

The advantages of persistence can be summarised as follows:

- Reduced complexity.
- Reduced code size and time to execute.
- Protection mechanisms that operate over the whole environment.
- Referential integrity preserved over the environment.

The first saving of persistent systems is in the reduced complexity for application builders. Traditionally the programmer has to maintain three mappings among the database model, the programming language model and the real world model of the application as can be seen in Figure 3.

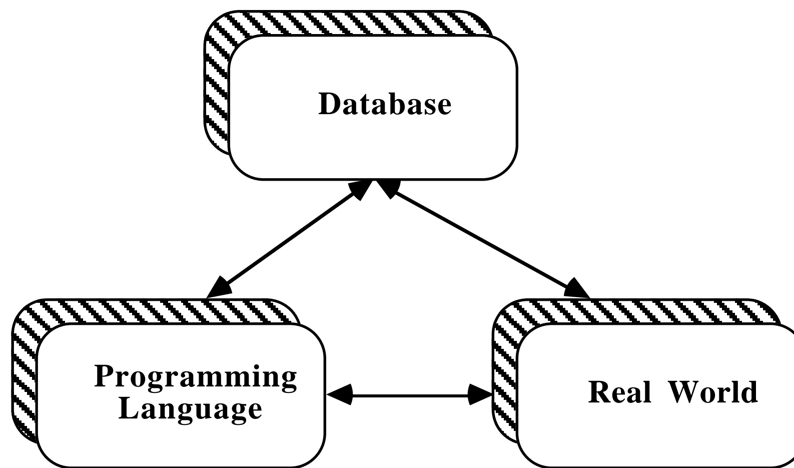

Figure 3: The mapping in traditional applications systems.

The intellectual effort in maintaining the mappings distracts the programmer from mastering the inherent complexity of the application to concentrate on overcoming the complexity of the support system. Figure 4 illustrates that in a persistent system the number of mappings is reduced from three to one thereby simplifying the application builders' task.

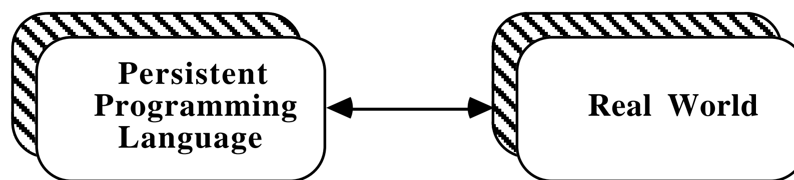

Figure 4: Mapping in a persistent system.

Corresponding to the intellectual savings of persistence there is also a saving in the amount of code required to maintain the mappings. This has been estimated at 30% of the total code for a typical database application [6]. The code that is unnecessary is concerned with the explicit movement of data between main and backing store and the code required to change the representation of the data for long term preservation and restoration. An example of the former is input and output code and of the latter is code to flatten and reconstruct a graph before and after, output and input respectively. Not only is the size of the application code reduced thereby producing savings throughout the software life cycle of the application but also the time to execute and store this code is saved in physical terms.

The third benefit of persistent systems is that a single model of protection may operate over the whole environment. In most programming languages the simplest way to break the type system is to output a value as one type and input it again as another. Thus the type security is lost over the persistent store. Using a single enforceable model of type reduces complexity while increasing the protection over many current systems [35].

The final advantage of persistence is that referential integrity is preserved over the environment. Figure 5 illustrates a data structure where one of the components is pointed at or shared by two others. This sharing should be preserved within the persistent store.

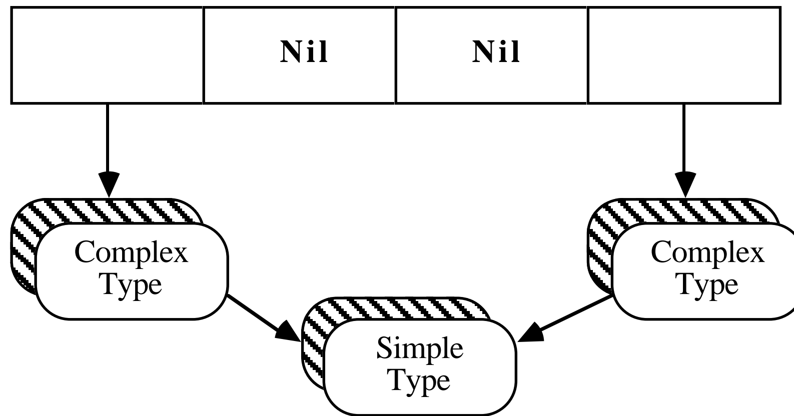

Figure 5: Sharing and referential integrity.

Figure 6 shows how referential integrity may be broken as in the persistent models of Amber [16] and a proposal for ML.

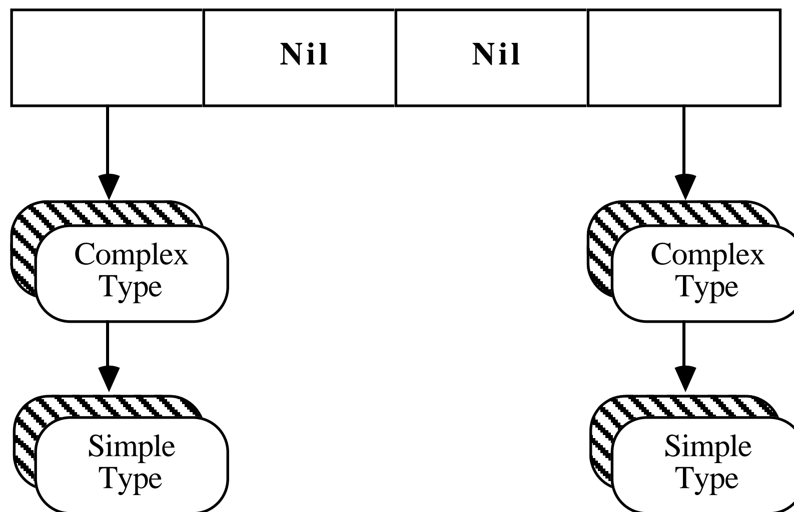

Figure 6: A lack of referential integrity.

2.4. Costs of Persistence

In common with all abstraction mechanisms there are costs in using and constructing persistent programming systems. The main costs over traditional systems are :

- The cost of constructing a stable object store.
- The loss of efficiency with some applications due to the abstraction.
- The cost of providing language independent binding mechanisms.

The cost of constructing a stable object store can vary according to the method used. Several schemes for extending the work of Lorie in using the virtual memory manager to identify and preserve modified data have been proposed [27,38,40,44,45]. Other techniques similar to

software segmentation are also common [5,13,14,18]. The cost of constructing the stable stores is considerable regardless of the method used.

There is a loss of efficiency in implementing some algorithms within a persistent store. This is due to the fact that the user does not have full control of all the physical properties of the machine. Since these are deliberately abstracted over, they may not be used. The situation is similar to that of a virtual memory system that does not allow users to perform their own paging.

Finally, there is a cost to be paid for language independent bindings. These mechanisms usually have to integrate disparate type systems which can be very expensive [10,31].

2.5. Myths

There are two myths about persistent systems that were first thought to be true. We now feel that the claims were premature. They are :

- persistence can be added to any language, and
- persistence can model all our computational needs efficiently.

The integration of a programming language with a persistent store requires some basic facilities within the language for success. A good example of an unsuitable host is the language Ada [26] where there is a clear distinction between program and data. Programs are kept in libraries and data in the file system. Furthermore input and output on **access** types is undefined. Both of these problems make it impossible to add persistence without redefining the language which in this case is forbidden.

The second myth is that persistent languages can model all our computation needs efficiently. It should be obvious that some algorithms, such as a disk sort, cannot even be implemented in orthogonally persistent systems.

2.6. Orthogonal Persistence Systems Requirements

Orthogonal persistence requires the following for support :

- Programming languages in which computations may be expressed using orthogonal persistence, and
- System architectures, both software and hardware, to support the implementation of these languages.

We will now discuss both of these.

3. PERSISTENT PROGRAMMING LANGUAGES

The provision of persistence should be orthogonal to all other aspects of programming language design. Therefore it should be possible to have applicative, relational, logic, object,

query and imperative persistent programming languages. Some known persistent programming languages are :

- Pascal/R [41] - relational,
- DBPL [29] - relational,
- PS-algol [36] - imperative,
- Leibniz [23] - object,
- E [37] - sort of object (C^{++}),
- Galileo [2] - object with extentional classes,
- Poly [30] - applicative,
- Staple [28] - applicative,
- Amber [16] - applicative,
- Persistent Prolog [19,24] - logic,
- χ [25] - capability,
- Napier88 [33] - store and multiparadigm.

Plus many other claims, counter claims and proposals! The status of each implementation can be obtained by asking for a copy of the system.

3.1. Challenges for Persistent Languages

The provision of an environmental model of computation does have an influence upon certain language features to support its holistic view of information. Most notably it must pay attention to the following aspects of language design :

- ***Name Space Control*** - to uniquely identify objects,
- ***Scale*** - to express computations on large bodies of data,
- ***Data Evolution*** - to control change including schema evolution,
- ***Sharing*** - to avoid re-invention of wheels,
- ***Protection*** - to ensure the integrity of valuable data,
- ***Concurrency*** - to allow parallel access,
- ***Transactions*** - to give a semantics to generalised update in a concurrent shared environment and provide a notion of stability, and

- ***Complexity Control*** - to avoid the failures of current technology and achieve simpler more cost effective solutions.

Thus nearly all aspects of language design have to be re-thought to accommodate the integrated view of data that persistence provides.

3.1.1. Name Space Control and Binding

For independently prepared programs and data to be correctly composed requires a binding mechanism [10,31]. We wish the following facilities :

- Re-useability and distribution of programs and data.
- The ability to combine data from many sources.
- Incremental program and data definition.
- The ability to identify data - how are names used?

The challenges in name space control and binding are :

- To find the balance between static and dynamic binding.
- To develop generic software architectures for applications' construction.
- To find language constructs to express flexible incremental binding (FIBS) to avoid forcing binding too early.

3.1.2. Scale

Persistent programming languages with their environmental view of data require the ability to manipulate large bodies of data. Large scale is a natural consequence of longevity but may also occur in automatic data collection. Scale is relative and becomes an issue when the size of the data approaches the limits of the technology. Today's large scale may be small by tomorrow's standards.

The challenges of scale are to invent :

- A method of describing large scale data and computations.
- Efficient storage and algorithms to manage access to the large scale data.
- Methods to ensure that data creep does not occur since that will eventually become unacceptable.

3.1.3. Data Evolution

Since the uses of the data cannot be predicted fully in advance a method of evolving existing data is required. It is unlikely that all data will change on every computation which means a method of incremental change is required. The changes to data have 3 categories :

- Changes in representation.
- Changes in organisation.
- Changes in the meta data.

The challenges of data evolution are :

- To find understandable semantics for change.
- Because of scale, change may require incremental algorithms for adequate performance.
- Limiting the propagation of change especially in the case of meta data evolution.

3.1.4. Sharing

A large body of data requires a community effort for its construction and maintenance. This requires the ability to share the data and gives rise to a number of other motivators. These are :

- Protection - to protect users from
    - the system
    - other users
    - themselves
- Concurrency - to allow parallel access
- Transactions - a model to provide
    - availability of data for access
    - a notion of stability

The challenge of sharing is to find and implement a *good* model of sharing

3.1.5. Protection

Large bodies of data are inherently valuable. It is essential to protect them from accidental and deliberate misuse either by software or hardware [35]. Among the mechanisms commonly used for protection are :

- Integrity constraints - used in database systems to ensure that the data is kept mutually consistent.
- Type systems - used in programming languages to control the interaction of objects.
- Stable storage - used to ensure that when failure does occur that the system can limit its losses due to component failure.
- Capability architectures to control access.

The challenges of protection are :

- To perform as much eager checking as possible.
- To design an adequate model of type for persistent programming.
- To design mechanisms for ensuring integrity & security.
- To design and engineer stable stores sufficient for our applications.

3.1.6. Concurrency and Transactions

Different models of both concurrency and transaction may have to be provided for persistent systems [32]. Long lived data has an impact on the choice of transactional model since serialisability may be too restrictive for the efficient sharing of data. We require at least the following :

- There should be mechanisms to permit and control shared and concurrent usage.
- It should be possible to encapsulate any sequence of operations on the store into a transaction and have it behave like a single operation.

The challenges of concurrency and transactions are :

- To implement concurrency and transaction models efficiently.
- To find models without serialisability.

3.1.7. Controlling complexity

Complexity in the development system causes the user to be distracted from the computational task to concentrate on mastering the implementation technology. Abstraction is a process that allows the general pattern to be observed while ignoring the inessential details. By supporting powerful abstraction facilities and keeping the development system simple, there is the possibility of concentrating on the complexity of the application. This allows more complex and more powerful systems to be developed.

The challenge of controlling complexity is to invent and implement appropriate abstractions for persistent programming

3.2. Type Systems for Persistent Programming

The long term goal of research into type systems is to develop an adequate model of type that meets the computational needs of persistent systems [3]. Ideally we would like a simple set of types, and a type algebra, so that by a succession of operations and the provision of parameters, any data model or conceptual data model can be defined [7].

Type systems provide two important facilities within both databases and programming languages, namely data modelling and data protection. Data modelling is performed in databases using data models and in programming languages by a classical type system. In the future the traditional database schema will be regarded as a type. Data protection is provided by integrity constraints in databases, type checking in programming languages and dynamic constraints such as capabilities in operating systems. All these mechanisms require integration into a coherent whole.

The issue of type checking is central to this activity. Static checking allows assertions to be made and even proved about a computation before it is executed. It therefore provides a level of safety within the system. Dynamic checking is however sometimes necessary for rebinding or merging of schema. The aim is to pursue the limits of static checking.

There are two approaches currently used to provide static checking for database type systems. The first is constraint specification where constraints over the data for a particular computation are expressed in some language. The checking requires a powerful theorem prover beyond the limits of those currently available. Such systems are usually undecidable and an unsuccessful check may be caused by the limitations of the theorem prover rather than inconsistent constraints. The second approach is to extend classical type systems. These describe decidable types with a simpler syntax which allows the user to better understand type checking failures. Both these approaches need to be pursued. Hopefully they will converge.

The major challenges for type systems for persistent programming are to provide the following:

- Polymorphism

  Polymorphism is essential to the expression of generic computations. Some known and well studied forms of polymorphism include parametric, inclusion, bounded quantification and subtyping [15,17,20,34]. Further forms and efficient implementations are required in the next generation of persistent systems.

- Bulk Types

  Persistent systems express computations on large scale data. The data may be aggregated in bulk data types such as relations, sets, lists and arrays and is

computed upon using query languages. One interesting innovation is the extentional notion of type that appears in OODBMS where a class is defined as the set of existing values of a particular type. Queries are made over the classes [9,12,47]. The theory of classes is not as yet well worked out and the semantics of update in queries over bulk types still not resolved. A further advance may also come from heterogeneous collections.

- View Mechanisms

  View mechanisms are used in database to allow users to concentrate on a subset of the data in a database. Such mechanisms are essential to the integrity of the database since a system administrator can use a view to restrict the access given one or a class of users. At present views are constructed by ad hoc methods and no underlying theory of views exists. Recently abstract data type have been proposed as one possible solution to this in the form of existentially quantified types [20]. This work is however in its infancy and needs more research to be convincing for large scale applications.

- Performance of Type Checkers

  The type of a database schema can become very large. This raises performance issues of how to best represent the type within the database. Also of concern is the performance of the type checking algorithm since it is used when persistent systems as schema are merged. Very little work has been done in this area.

- Object Migration

  Can objects change their type?. For example, can a person become a student for a period of a computation? This is clearly desirable in modelling terms and has to be better understood in type checking terms.

- Schema Evolution

  A type system for a persistent programming language requires a mechanism to provide modularity, storage and reuse of type definitions for incremental schema evolution. Schema evolution may be provided by using the reflective facilities of the next section.

3.3. Reflection

Reflection is the ability of a system to support its own evolution. This may entail changes to the data, the programs that manipulate the data and the schema. Reflection is of great interest in persistent systems since they require the ability to evolve. It yields an extreme form of dynamic binding. Given that a language is Turing complete, reflection cuts out a level of interpretation that would be required to provide the extra layer of expressibility to support total evolution.

Reflection can be used to provide the following :

- genericity [42],
- browsing [22],
- schema evolution [22],

- querying,
- data models,
- adding types to the value space [22].

Work on reflection in persistent systems is as yet not well developed.

4. PERSISTENT ARCHITECTURES

Persistence poses the same set of problems for architectures as it does for languages [8,11,21]. However, whereas the persistent languages are concerned with the expression of computations using the persistence abstraction, the architectures are concerned with the efficient implementation of these languages. Given that the technologies for the efficient translation of languages is well understood we will concentrate on the technology for the implementation of the object store.

4.1. Intrinsic Properties of the Persistent Store

The properties of a persistent store can be divided into intrinsic and technological. The intrinsic properties are those that are desirable for the persistent store to possess whereas the technological properties arise in trying to implement the intrinsic properties.

The intrinsic properties are:

- a freedom in binding mechanisms that allows the reuse of components in the store to match the needs of the particular application,
- unbounded capacity to match the conceptually unbounded nature of the stores provided in some languages,
- infinite speed, and
- error free.

The above intrinsic properties of persistent stores are not realistic in terms of implementation with current technology. They therefore have to be approximated in such a manner that the persistence abstraction is not totally compromised. In particular, a gross violation would occur if the user is required to write code to support the movement of data.

4.1.1. Binding Mechanisms

The intrinsic problems of binding are identifying persistent data and avoiding binding too early. Persistent data is normally identified by computing the transitive closure of data from some root. The technological problem is how to perform this efficiently. For example, it should be possible to compute the transitive closure without bringing all the data into the main store which is very expensive in terms of disk transfers.

For efficient implementation, current technology requires binding that is too early. The assumption in most current systems is that a persistent store will be populated with a large

number of statically bound objects that can be dynamically bound into an application invocation. The fewer dynamic binds that are performed the more efficient the system. However, static binding restricts the flexibility of using the components of the statically bound objects.

The early binding problem occurs at two levels. In order to build an efficient persistent store some assumptions are made about the pattern of use of the store. This constitutes a binding at the system construction time. Two reasons for this are the efficient management of address bits and efficient garbage collection. However little is known about the use of persistent stores and the early assumptions may be invalid over a long period of time.

The second manifestation of early binding occurs at the data modelling level. For example the object-oriented model binds data within objects for the lifetime of the object. The problem is that static binding in a persistent store is forever and if it is subsequently discovered that the binding is inappropriate then the user is stuck.

We require binding that will allow the reuse of components appropriate to the applications needs - not the needs of the architecture. For example O_2 [12] queries look inside objects as do the views of Napier88 [33]. The group and ungroup options in MacDraw allows the freeing and rebinding of objects and is a hint as to how both static and dynamic binding may be accommodated together in persistent systems.

4.1.2. Unbounded Capacity

The intrinsic property of a store of unbounded capacity is how to name objects within it. A flat name space or a conceptual name space can be used. The technological properties are :

- Movement of data to the active store.
- Movement of data to the long term store.
- Management of very large values.
- Management of very large stores.
    - very large address space - Monads [1,39], System 38 [45].
    - extensible contextual addressing.
    - distribution and coherence of data.
    - copying, detaching and merging stores.
    - garbage collection.
    - store fragmentation.
    - very large physical stores

4.1.3. Infinite Speed

There are only technological properties of systems trying to simulate infinite speed. Concurrency mechanisms are used in both modelling data and for improving the performance of the overall system. At an architectural level this involves multiprocessors if real concurrency is to be achieved. In turn this poses the question of whether distribution is and appropriate language and architectural model. Both store and processor distribution are possible. Finally system wide locking has to be provided to ensure the integrity of data and allow transactions to be implemented.

4.1.4. Error Free Stores

A persistent store should be free from unexpected failures of hardware and software. The intrinsic problem is the semantics of failure. That is, how is failure explained to the user

The technological problems centre around the following

- Stability mechanisms
  - implicit stability - e.g. shadow paging ...
  - explicit stability - e.g. transaction logs ...
- Protection mechanisms.
  - integrity constraints.
  - type checking - static and dynamic.
  - capabilities - software and hardware.

4.1.4.1. Questions of Stability

There are some technological properties associated with stability alone. They are:

- What percentage of the data is required to automatically survive a failure of any or all of the equipment at a single site?
- Is some data more valuable than other data?
- What percentage of the data is required to be restorable?
- How frequently should data be written to stable store?
- What processing and equipment overhead is acceptable for stability?
- How quickly must data be restored?
- How are restorations performed? - interactively to avoid cascading?
- How is restoration explained to the user?

5. THE ULTIMATE CHALLENGE

To build a persistent system the high ideals should be adhered as closely as possible. The particular instance of the architecture yields an efficient implementation for a particular class of problems. It will almost certainly be general purpose but not the most efficient for all applications.

Most persistent programming systems are still small scale. The largest PS-algol system is about 200 megabytes (2.5 million objects) in a company that expects serious users to have between 1 to 10 databases of about 10 to 20 gigabytes.

We need a series of large experiments to refute or verify all the ideas and concepts.

ACKNOWLEDGEMENTS

The work was supported by ESPRIT II Basic Research Action 3070 - FIDE.

REFERENCES

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

2. Albano A., Cardelli L. & Orsini R. "Galileo: A Strongly Typed, Interactive Conceptual Language." *ACM Transactions on Database Systems*, vol. 10, no. 2, 1985, pp. 230-260.

3. Albano, A., Dearle, A., Ghelli, G., Marlin, C., Morrison, R., Orsini, R & Stemple, D. "A Framework for Comparing Type Systems for Database Programming Languages". *2nd International Workshop on Database Programming Languages*, Oregon (1989), pp. 203-212.

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

5. Atkinson M.P., Chisholm K.J. & Cockshott W.P. "CMS - A Chunk Management System." *Software Practice and Experience*, vol. 13, no. 3, (1983), pp. 259-272.

6. 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, pp. 360-365.

7. Atkinson, M.P. & Morrison, R. "Integrated Persistent Programming Systems". *19th International Conference on System Sciences*, Hawaii, U.S.A., (January 1986), pp. 842-854.

8. Atkinson, M.P., Morrison, R. & Pratten, G.D. "Designing a persistent information space architecture". *10th IFIP World Congress*, Dublin (September 1986), pp. 115-120.

9. Atkinson, M.P. & Morrison, R. "Polymorphic Names, Types, Constancy and Magic in a Type Secure Persistent Object Store". *2nd International Workshop on Persistent Object Systems*, Appin, (August 1987), pp. 1-12.

10. Atkinson, M.P., Buneman, O.P. & Morrison, R. "Binding and Type Checking in Database Programming Languages". *The Computer Journal*. 31,2 (1988), pp. 99-109.

11. Bancilhon F., Barbedette G., Benzaken V., Delobel C., Gamerman S., Lecluse C., Pfeffer P., Richard P. & Valez F. "The Design and Implementation of O_2, an Object Oriented Database System". Proc. 2nd International Workshop on Object-Oriented Database Systems, West Germany. In *Lecture Notes in Computer Science, 334.* Springer-Verlag (September 1988), pp. 1-22.

12. Bancilhon, F., Cluet, S. & Delobel, C. "A Query Language for the O_2 Object-Oriented Database". *2nd International Workshop on Database Programming Languages*, Salishan, Oregon (1989).

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

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

15. Cardelli, L. & Wegner, P. "On understanding types, data abstraction and polymorphism". ACM *Computing Surveys* 17, 4, (December 1985), pp. 471-523.

16. Cardelli, L. *Amber*. Tech. Report AT7T. Bell Labs. Murray Hill, U.S.A. (1985).

17. Cardelli, L. "Typeful Programming". *DEC SRC Report*, (May 1989).

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

19. Coloumb, R.M. "Issues in the Implementation of Persistent Prolog". *Proceeding of the 3rd International Workshop on Persistent Object Stores,* Newcastle, N.S.W. (Jan 1989), pp. 67-79.

20. Connor, R.C.H., Dearle, A., Morrison, R. & Brown, A.L. "Existentially Quantified Types as a Database Viewing Mechanism". *Advances in Database Technology - EDBT90,* Venice. In **Lecture Notes in Computer Science**. 416. Springer-Verlag (1990), pp. 301-315.

21. Connors T. & Lyngbaek P. "Providing Uniform Access to Heterogenous Information Bases". Proc. 2nd International Workshop on Object-Oriented Database Systems, West Germany. *In Lecture Notes in Computer Science, 334.* Springer-Verlag, (September 1988), pp. 162-173 .

22. Dearle A. & Brown A.L. "Safe Browsing in a Strongly Typed Persistent Environment". *The Computer Journal* 31,6, (December 1988), pp. 540-545.

23 Evered, M. "Leibniz - A Language to Support Software Engineering". PhD thesis Technical University of Darmstadt (1985).

24. Gray, P.M.D., Moffat, D.S. & Du Boulay, J.B.H. "Persistent Prolog: A Searching Storage Manager for Prolog". *Proceedings of the 1st International Workshop on Persistent Objects Systems,* Appin, Scotland (August 1985), PPRR-16-85, Universities of Glasgow and St Andrews, pp. 353-368.

25. Hurst, A.J. & Sajeev, A.S.M. "A Capability Based Language for Persistent Programming: Implementation Issuse". *Proceeding of the 3rd International Workshop on Persistent Object Stores*, Newcastle, N.S.W. (Jan 1989), pp. 186-201

26. Ichbiah et al., The Programming Language Ada Reference Manual. ANSI/MIL-STD-1815A-1983. (1983).

27. Lorie A.L. "Physical Integrity in a Large Segmented Database." *ACM Transactions on Database Systems*, vol. 2, no. 1, 1977, pp. 91-104.

28. McNally, D.J. "Code Generating Functional Language Modules for a Persistent Object Store". University of St Andrews STAPLE/89/2 (1989).

29. Matthes, F. & Schmidt, J.W. "The Type System of DBPL". Proceeding of the 2nd International Workshop on Database Programming Languages, Salishan, Oregon (June 1989), pp. 219-225.

30. Matthews, D.C.J. "Poly Manual" Technical Report 65, University of Cambridge, U.K. (1985).

31. Morrison, R., Brown, A.L., Dearle, A. & Atkinson, M.P. "Flexible Incremental Binding in a Persistent Object Store". *ACM.Sigplan Notices*, 23, 4 (April 1988), pp. 27-34.

32. Morrison, R., Brown, A.L., Carrick, R., Connor, R.C. & Dearle, A. "On the Integration of Object-Oriented and Process-Oriented Computation in Persistent Environments". Proc. 2nd International Workshop on Object-Oriented Database Systems, West Germany (1988). In **Lecture Notes in Computer Science**, 334. Springer-Verlag, (September 1988), pp. 334-339.

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

34. Morrison R., Dearle A, Connor R.C.H. & Brown A.L. "An ad hoc Approach to the Implementation of Polymorphism". University of St Andrews CS/90/1 (1990).

35. Morrison R., Brown A.L., Connor R.C.H., Cutts, Q.I., Dearle, A., Kirby, G., Rosenberg J.,& Stemple, D.., & Munro D.S. "Protection in Persistent Object Systems" *International Workshop on Computer Architectures to Support Security and Persistence*, Universität Bremen, West Germany, (May 1990).

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

37. Richardson, J.E. & Carey, M.J. "Implemeting Persistence in E".. *Proceeding of the 3rd International Workshop on Persistent Object Stores*, Newcastle, N.S.W. (Jan 1989), pp. 302-319.

38. 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).

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

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

41. Schmidt, J.W. "Some high level language constructs for data of type relation". *ACM.TODS* 2, 3 (1977), pp. 247-261.

42. Stemple, D., Fegaras, L., Sheard, T. & Socorro, A. "Exceeding the Limits of Polymorphism in Database Programming Languages". *Advances in Database Technology - EDBT90,* Venice. In **Lecture Notes in Computer Science**. 416. Springer-Verlag (1990), pp. 269-285.

43. Stroustrup B. *The C++ Programming Language*, Addison Wesley (1986).

44. 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), pp. 148-159.

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

46. Wileden, J.C., Wolf, A.L., Fisher, C.D. & Tarr, P.T. "PGraphite: An experiment in Persistent Typed Object Management". *SIGSOFT88* (1988), pp. 130-142.

47. Zdonik, S.B. "Query Optimisation in Object-Oriented Databases". *22nd Hawaii International Conference on System Sciences* (Jan 1989), II, pp. 19-25.