Implementation of Distributed Orthogonal Persistence Using Virtual Memory

Francis Vaughan

Thesis submitted for the degree of
Doctor of Philosophy
in
The University of Adelaide
(Faculty of Mathematical and Computer Sciences)

December 1994

Awarded 1995

This work contains no material which has been accepted for the award of any other degree or diploma in any university or other tertiary institution and, to the best of my knowledge and belief, contains no material previously published or written by another person, except where due reference has been made in the text.

I give consent to this copy of my thesis, when deposited in the University Library, being available for loan and photocopying.

SIGNED: **DATE:** 17th August 1995

Abstract

Persistent object systems greatly simplify programming tasks, since they hide the traditional distinction between short-term and long-term storage from the applications programmer. As a result, the programmer can operate at a level of abstraction in which short-term and long-term data are treated uniformly. In the past most persistent systems have been constructed above conventional operating systems and have not supported any form of distributed programming paradigm. In this thesis we explore the implementation of orthogonally persistent systems that make direct use of the attributes of paged virtual memory found in the majority of conventional computing platforms. These attributes are exploited to support object movement for persistent storage to addressable memory, to aid in garbage collection, to provide the illusion of larger storage spaces than the underlying architecture allows, and to provide distribution of the persistent system.

The thesis further explores the different models of distribution, notably a one world model in which a single persistent space exists, and a federated one in which many co-operating spaces exist. It explores communication mechanisms between federated spaces and the problems of maintaining consistency between separate persistent spaces in a manner which ensures both a reliable and resilient computational environment. In particular characterising the interdependencies using vector clocks and the manner in which vector time can be used to provide a complete mechanism for ensuring reliable and resilient computation.

The thesis concludes with a description of a new operating system design in which support for the mechanisms described earlier are intrinsic in the design. This operating system is able to provide orthogonal persistence in a distributed environment with no effort on the part of users of the operating system.

Acknowledgments

This thesis owes much to many people.

First and foremost thanks go to my parents for their love, support and forbearance over the last few years.

Thanks are due to my supervisor Al Dearle, to whom much is owed, especially for his friendship and encouragement in this endeavour.

Ron Morrison and the persistence group at the University of St Andrews: Richard Conner, Quintin Cutts, Graham Kirby and Dave Munro for advice, encouragement and a never wavering welcome when I landed upon their doorstep.

John Rosenberg and the Grasshopper group at Sydney University: Franz Henskens, Anders Lindström, Rex di Bona, Matty Farrow and Stephen Norris, for their part in the Grasshopper project. Also for the stimulating and rowdy discussions which made the Grasshopper project such a joy to be a part of.

Alex Farkas and David Hulse for their respective contributions to Casper and Grasshopper.

The original members of the Casper group, Tracy Lo Basso, Ruth Fazakerley and Bett Koch for putting up with me.

Chris Barter for support and friendship over many years, without which this would not have been possible.

The members the department at Adelaide University, for providing a convivial home from home for so many years.

Contents

Chapter 1. Introduction.

1.1. Persistence 1
1.2. Accidents of History 4
1.3. Models of persistence 5
- 1.3.1. Specially designated objects 5
- 1.3.2. Embedded persistence 7
- 1.3.3. Reachability 7

1.4. Distribution 8
- 1.4.1. Single Unstructured 8
- 1.4.2. Structured single address space 9
- 1.4.3. Fully partitioned 9
  - 1.4.3.1. Federated 10
- 1.4.4. Difficulties 10

1.5. Implementation Mechanisms 11
- 1.5.1. Specialised Hardware 11
- 1.5.2. Software 11
- 1.5.3. Conventional Hardware 12
  - 1.5.3.1. Granularity 12
  - 1.5.3.2. Native code support 13

1.6. The Thesis 13
- 1.6.1. Contributions. 14

Chapter 2. Implementation Tactics.

2.1. Introduction 17
2.2. Issues 17
- 2.2.1. Data Movement 18
  - 2.2.1.1. Snapshots, Boot and Resume, and Undump. 18
  - 2.2.1.2. Data Movement in Persistent Systems 19
- 2.2.2. Address Generation 19
- 2.2.3. Pointer Representation. 20
- 2.2.4. Garbage Collection 20
  - 2.2.4.1. Generation GC 21
  - 2.2.4.2. GC in Persistent Systems 22
- 2.2.5. Basic architectures. 22
  - 2.2.5.1. Object Tables 23

2.3. Hardware 24
- 2.3.1. Monads 25
- 2.3.2. Lisp Machines 26
  - 2.3.2.1. Data Format 26
  - 2.3.2.2. Address Generation and Data Movement 27
  - 2.3.2.3. Garbage Collection 27
- 2.3.3. ORSLA 28
  - 2.3.3.1. Data Formats 28
  - 2.3.3.2. Garbage Collection 29
  - 2.3.3.3. Data Movement 30
- 2.3.4. Rekursiv 30
  - 2.3.4.1. Data Movement 31
  - 2.3.4.2. Object Format 32
  - 2.3.4.3. Garbage Collection 33

2.4. Software 33
- 2.4.1. Smalltalk 34
  - 2.4.1.1. Recognising Pointers 35
- 2.4.2. LOOM 36
  - 2.4.2.1. Smalltalk and Mneme 38
  - 2.4.2.2. Garbage Collection 41
- 2.4.3. PS-algol and CPOMS 42
  - 2.4.3.1. Computational model 42

2.4.3.2. ....Addresses in PS-algol and CPOMS 42
2.4.3.3. ....Data Movement 43
2.4.3.4. ....Object Formats and Identifying Pointers 44
2.4.3.5. ....Garbage Collection 45
2.4.4. ....Napier-88 and the PAM 45
2.4.4.1. ....PAM 46
2.4.5. ....The POMS Store. 47
2.4.6. ....Object Format 47
2.4.6.1. ....Data Movement 48
2.4.6.2. ....Address Generation 49
2.4.6.3. ....Garbage Collection 49
2.5. ....Page Based 50
2.5.1. ....Data Movement 51
2.5.1.1. ....File Mapping 52
2.5.2. ....Texas and ObjectStore 52
2.5.2.1. ....ObjectStore 53
2.5.2.1.1..................Data Movement in ObjectStore 53
2.5.2.1.2..................Garbage Collection 54
2.5.2.2. ....Texas 54
2.5.2.2.1..................Data Movement 54
2.5.2.2.2..................Pointer formats 55
2.5.2.2.3..................Address Creation. 55
2.6. ....Casper 55
2.6.1 ....Integrated Store Management 55
2.6.2. ....Data Movement 55
2.6.3. ....Object Formats and Pointer Detection 56
2.6.4. ....Address Generation 59
2.6.5. ....Local Heaps and Pointer Quarantine 59
2.6.5.1. ....Distributed Casper 59
2.6.5.2. ....Copy Out Implementation 60
2.6.5.3. ....Eager Copy-Out 60
2.6.5.4. ....Lazy Copy-Out 61
2.6.5.5. ....Page/Card Based 62
2.6.6. ....Store level garbage collection. 62
2.6.6.1. ....Opportunistic memory recovery 63
2.6.6.2. ....Full Garbage Collection 63
2.6.6.2.1..................Mark Phase 64
2.6.6.2.2..................Translation Phase 64
2.6.6.2.3..................Compaction 64
2.6.6.3. ....Summary of Garbage Collection 65
2.7. ....Performance 65
2.7.1. ....Cost of exceptions 66
2.7.1.1. ....Exceptions with CISC architectures. 66
2.7.1.2. ....Exceptions in Berkeley RISC. 67
2.7.1.3. ....Exceptions on conventional
....RISC architectures 67
2.7.1.4. ....Exceptions in Unix implementations. 68
2.7.1.5. ....Exceptions in OSF and Mach. 68
2.7.1.6. ....Measuring Exception Cost 68
2.7.1.7. ....Building Exceptions for Speed 69
2.7.2. ....Page Granularity 70
2.7.2.1. ....Page cards vs. remembered sets: 70
2.7.2.2. ....Object Residency Test 71
2.8. ....Comparisons. 72
2.8.1. ....Notes. 75

Chapter 3. Pointer Swizzling.

3.1. ....Introduction 77
3.2. ....Software address translation 78
3.3. ....Address translation at page fault time 80

3.4. A hybrid approach 82
3.4.1. Exceptions 84
3.4.2. Data Load. 85
3.4.3. Eager Swizzling 86
3.4.4. Deswizzling 88
3.4.4.1. Deswizzling in Wilson's Scheme 90
3.4.4.2. Deswizzling in the Hybrid Scheme 90
3.4.5. Elaboration of detail 91
3.4.5.1. Finding object addresses 91
3.4.5.2. Pointer comparisons 96
3.4.5.3. Large Objects 97
3.4.5.4. Management of the translation table 97
3.4.5.5. Creation of new objects 97
3.5. Comparison of the schemes 98
3.6. Conclusions 100

Chapter 4. Store Architecures.

4.1. Introduction 101
4.2. Basics 101
4.2.1. Size 102
4.2.2. Speed. 102
4.2.3. Reliability. 102
4.3. Stability and Resilience 102
4.3.1. Soft Failure 103
4.3.2. Hard failure 103
4.3.3. Site Failure 105
4.4. System reference model 105
4.4.1. Self Consistency 107
4.4.2. Synchronous and Asynchronous Creation of Persistent State. 107
4.4.3. Modelling. 108
4.5. Mechanisms 109
4.5.1. Snapshots 109
4.5.2. Incremental baseline generation 109
4.5.3. Logging. 110
4.5.3.1. Modelling Logging. 110
4.5.3.2. Undo and Redo in the one store. 112
4.5.3.3. Recovering logs 112
4.5.3.4. Replay Logging 113
4.5.4. Nested Store Strategies 114
4.5.5. Output in replay systems 114
4.6. Concurent Access, Distributed Systems and Multiple stores. 115
4.6.1. Causality Tracking 116
4.6.2. Replay of Output. 116
4.6.3. Exploiting Log Granularity 117
4.7. Trade-offs 118
4.8. Implementation Strategies. 118
4.8.1. Challis's Algorithm 119
4.8.2. Shadowing 120
4.8.2.1. After-Look 120
4.8.2.2. Before-Look 121
4.8.2.3. Shadow Paging 121
4.8.2.5. Maintaining locality 122
4.8.3. Object Shadowing 123
4.9. Conclusions 123

Chapter 5. Implemetation Strategies.

5.1. ....Introduction 125
5.2. ....Related Work. 125
5.2.1......................................Shrines 125
5.2.2......................................Page Based Napier88 Stores 125
5.2.3......................................Browns Before-Look Store 125
5.2.3.1. ....Operation 126
5.2.4......................................Munro's After-Look Store 127
5.2.5......................................Thatte 129
5.2.6......................................Logging Systems 131
5.2.7......................................RVM 131
5.2.8......................................Texas 132
5.2.9......................................Mneme 133
5.3. ....CASPER Bi-Phase 135
5.3.1......................................Basics 135
5.3.2......................................Page Map. 136
5.3.3......................................Store Structure 137
5.3.4......................................Store Creation 139
5.3.5......................................Secondary Files 140
5.3.6......................................Store Start-up 140
5.3.7......................................Napier88 Implementation 141
5.3.8......................................Runtime Actions 142
5.3.8.1. ....Page States 142
5.3.8.2. ....Read Access 142
5.3.8.3. ....Write Access 144
5.3.8.4. ....LPMap mapping 145
5.3.9......................................Meld 146
5.3.10....................................Recovery 147
5.3.11....................................Implementation Specifics 148
5.3.11.1. ....Placement in memory 148
5.3.11.2. ....Page copy sequence 148
5.3.11.3. ....Memory Allocation and Swap Space. 149
5.3.11.4. ....Implementation under Mach 149
5.3.11.4.1...............External Pager 150
5.3.11.4.2...............Problems with Mach 151
5.3.12....................................Partial State Meld 152
5.3.13....................................Multi-Phase 154
5.3.13.1. ....Store Modifications 155
5.3.13.2. ....Start-up 155
5.3.13.3. ....Normal Running 156
5.3.13.4. ....Meld Protocol 156
5.3.13.5. ....Snapshot Deletion 157
5.4. ....Comparisons and Conclusions 158
5.4.1......................................Log structured versus shadowing 158

Chapter 6. Distributed Casper.

6.1. ....Introduction. 161
6.2. ....Execution Environment. 161
6.2.1......................................Stability and Coherency 162
6.3. ....Overview of the Architecture 162
6.4. ....Stable store server 164
6.4.1......................................Store Stability 164
6.4.2......................................Associations 165
6.4.3......................................Stabilistion 168
6.4.4......................................Stable Store Heap Management 169
6.5. ....Clients 170
6.5.1......................................External Pager 170
6.5.2......................................Atomic Access 172
6.5.3......................................Local Heap Management 173
6.6. ....Cache coherency 174

6.6.1. Client Finite State Automaton 175
6.6.2. Stable Store Server Finite State Automaton 181
6.6.3. Interaction between the Automata 186
6.6.4. Omissions from the Simplified Automata 190
6.7. Conclusions 191

Chapter 7. Grasshopper.
7.1. Introduction 193
7.2. Why a new operating system? 193
7.3. Why conventional Hardware? 194
7.4. Grasshopper 195
7.4.1. Containers 195
7.4.2. Capabilities and Protection 196
7.4.3. Loci 196
7.4.4. Container mappings 197
7.4.5. Managers 198
7.5. Resilience 199
7.5.1. Causality 200
7.5.2. Lamport Time 202
7.5.3. Vector Time 203
7.5.4. Vector time in Grasshopper 205
7.5.4.1. Mapping 206
7.5.5. Stabilisation 207
7.6. Maintaining Global Consistency 208
7.7. Stability and Resilience 209
7.7.1. Initiation 209
7.7.2. Snapshotting State 210
7.7.3. Co-ordination 211
7.7.4. Progress 212
7.7.5. Failure Recovery 213
7.8. Logging, Determinism and Proxies 213
7.8.1. Deterministic Execution 214
7.8.1.1. Proxies 215
7.8.1.2. Concurrency 215
7.8.1.3. Mapping 216
7.9. Distribution 217
7.10. Putting it Together 218
7.10.1. Simple programs 218
7.10.2. Persistent Languages 219
7.11. Conclusions 219

Chapter 8. Conclusions
8.1. Summary 225
8.2. Page Based Systems 225
8.3. Tactics for Language Level Presentation 226
8.3.1. Data movement 227
8.3.2. Swizzling and Addressing 227
8.3.3. Pointer formats 227
8.3.4. Pointer Quarantine for Distribution and Garbage Collection 228
8.3.5. Persistence Orthogonal to native code 228
8.4. Store Design 229
8.5. Distribution 230
8.5.1. Distributed Casper 231
8.5.2. Grasshopper 231
8.6. Conclusions. 232
8.6.1. Status 232
Appendix A. 234
References 239

Chapter 1. Introduction

Orthogonal persistence allows all data to be treated in a manner independant of its lifetime. This thesis is about the implementation technology of persistent systems. Two interacting aspects of persistent systems are attacked. These are: the utilisation of the paged memory hardware provided in conventional virtual memory architectures, and distribution of persistent systems across separate but connected machines.

1. Persistence

The notion of persistence as a separate notion in computer systems was introduced by Malcom Atkinson [Atkinson 1978]. Persistence is defined as the length of time for which data both exists and is useable.

Persistence of data can be categorised as follows [Atkinson, Bailey et al. 1983]:

- data that only exists with the evaluation of an expression,
- data that is local to a procedure instantiation,
- data global to a program or that outlives the procedure instantiation that created it,
- data that exists between instantiations of a program,
- data that outlives the program that created it, and
- data that outlives the creator program in all its versions.

Conventional system have managed these various forms of persistence using different techniques. Typically the first three are under the control of the programming language, or more accurately the language compiler. The forth and fifth points have usually been taken care of by operating system controlled storage (typically files) or separate database systems. The last point is often not adequately addressed in many systems at all, indeed the problem of managing legacy systems and data is becoming more acute as time goes on.

By abstracting the notion of persistence away from any other attributes of data, the need to explicitly take notice of, and program according to, these attributes of persistence, is avoided. In particular, the need to take notice of the technologically imposed differences between fast but volatile, and long term but slow storage, is avoided. A system which completely abstracts the notion of persistence away from other visible aspects of data is termed *orthogonally persistent.*

The two basic principles behind orthogonal persistence are:

- that any object may persist for as long, or as short, a period as the object is required, and
- that objects may be manipulated in the same manner regardless of this longevity.

The requirements of a system which supports orthogonal persistence can be summarised as follows.

- Uniform treatment of data structures.

  Conventional programming systems require the programmer to translate data resident in virtual memory into a format suitable for long term storage. For example, graph structures must be translated into a flattened form when they are written into files or mapped into relations for storage in a conventional database. These activities are both complex and error prone. Persistent systems free the programmer from such encumbrances since data of any type with arbitrary longevity is supported by the system.

- Location independence.

  To achieve location independence, data must be accessed in a uniform manner, regardless of the location of that data. In distributed persistent systems, location independence may be extended to the entire computing environment by permitting data resident on other machines to be addressed in the same manner as local data [Koch, Schunke et al. 1990; Tam, Smith et al. 1990; Vaughan, Schunke et al. 1990; Henskens, Rosenberg et al. 1991; Henskens 1992; Vaughan, Schunke et al. 1992].

- Data resilience.

  All systems containing long-lived data must provide a degree of resilience against failure. Persistent systems must prevent the data stored in them becoming corrupt should a failure occur. In addition, since one of the goals of persistence is to abstract over storage, resilience mechanisms should not be visible at the user level.

- Protection of data.

  In any significant application some mechanism must be provided to protect data from accidental or malicious misuse. Without such protection execution of processes within the persistent system could result in erroneous processes corrupting data owned by other users. In persistent systems this is typically

provided through: the programming language type system [Morrison, Brown et al. 1990], data encapsulation [Liskov and Zilles 1974], capabilities [Fabry 1974], or by a combination of these techniques.

To date, most persistent systems, with a few exceptions [Rosenberg and Abramson 1985; Campbell, Johnston et al. 1987; Dasgupta, LeBlanc et al. 1988] have been constructed above conventional operating systems. Unfortunately these efforts are usually compromised by a mismatch between the abstractions provided by the host operating system and those which would more naturally be useful in the implementation of a persistent system. Two terms have been coined to express the most frustrating problems encountered. These are:

- Impedance mismatch [Bancilhon and Maier 1989].
- Semantic Gap.

Impedance mismatch is an evocative term derived from engineering theory. Power transfer systems are unable to transmit all the power available if the transmitter and receiver have mismatched impedances, this is independent of the intrinsic power handling capabilities of either. So too in the movement of data, systems in which there is a mismatch in the data access and storage mechanisms (for instance, access to random small objects for an object store in combintion with a stream based I/O model such as provided by Unix) result in poor performance, despite highly optimised implementations of each data transfer system.

The semantic gap is a related term. Rather than relating to performance directly it describes the difficulties and added burden shouldered by programmers when the sematic models of data storage to not mesh. For instance a conventional database may represent all data as relations. Such a model may be highly effective (and very efficiently implemented) for the target users. However considerable effort is required to make such a storage mechanism useful for more advanced programming paradigms, or simply for applications in which a much less structured view of data is used.

Faced with such sematic mismatches, the implementors of persistent languages or their support systems are invariably forced to construct an abstract machine above the abstraction of the host operating system. Usually such efforts are unable to make much headway in attacking the impedance mismatch. Indeed the extra layering of software is an impediment to performance. Performance is unlikely to be anything close to the intrinsic performance of the hardware unless the implementors are afforded access to the lowest

levels of the host system. When such access is granted they can attempt to build appropriate abstractions, avoiding as much of the inappropriate design as possible. Recently one of the areas in which such access has been allowed, and exploited, is in use of virtual memory mechanisms. This thesis explores use of virtual memory as an implementation tool in considerable detail.

2. Accidents of History

Current conventional computer systems have institutionalised the break between volatile short term data storage and long term permanent storage. That such a division should have occurred is arguably not intrinsic to the nature of computer system design. Indeed such a break may be regarded as being the result of a series of historical accidents.

Some of the very early systems developed held within them most of the attributes of orthogonal persistence. Designs using magnetic core memory were capable of retaining all data (including, in many cases, register values) in the face of system failures or shutdown. Indeed these designs are still in use in highly critical applications. However, the applications supported were not very sophisticated and the machine themselves so restricted in available storage that little use could be made of the intrinsic freedom potentially available.

The Atlas machine [Fotheringham 1961; T Kilburn 1962] was possibly one of the most influential designs, and offered one of the first architectures in which integration of volatile and stable storage was provided. The Atlas allowed a running program to generate addresses within a one mega-word address space. This space was partitioned into pages, each of 512 words in length. A special hardware translation table, with one entry per page of physical memory, mapped each generated address onto either physical memory or caused a trap which caused operating system code to load the missing data into physical memory, alter the translation table and allow the access to continue. Thus the programmer was able to view the available stable storage as simply an extension of the machine address space. The architecture was however limited by the need for an entry in the translation table for each page of physical memory, making extending the architecture expensive.

The lasting contribution to computer system design was unfortunately not the notion of the single level store, but rather the development of demand paged virtual memory, in which rather then making stable storage an extension of the program address space, stable storage is used to extend the volatile address space of the program. A subtle but crucial distinction.

Later architectures, in particular the IBM systems 38 [Bertis, Truxal et al. 1978], RS6000 and AS400 [Malhorta and Munroe 1992] did carry the notion of one level store further and integrated stable storage into an extended segmented addressing architecture.

3. Models of persistence

In general persistence is manifested to the user in three ways. These are:

- Specific designation of data objects. The programmer must designate persistent data objects through some language or library function.
- Embedded persistence. The entire program and data space is embeded within a persistent address space and is thus persistent.
- Reachability. Data items which are live by virtue of being reachable by programs in the persistent systems are maintained on stble storeage.

3.1. Specially designated objects

Many systems provide a measure of persistence by providing some mechanism by which the programmer can designate a datum as being persistent. The Amber language [Cardelli 1985] allows the programmer to inject a datum into a special data type (called *dynamic*). A pair of special commands, *export* and *import,* save and restore individual data objects to and from files. However the model does not preserve referential integrity, each time a data object is imported, a new copy is made.

A different mechanism is often used in languages which provide some inheritance mechanism in the type system. By inheriting from a "persistence" class, a data object includes appropriate mechanisms to allow it to move to and from stable storage. The E language [Richardson and Carey 1989] is an early example of this. Of particular note are a large number of systems designed using the C++ language.

The C++ language [Stroustrup 1986] has engendered considerable interest since its inception. A considerable number of implementations of systems that either provide for a "persistent C++" or provide a C++ class library through which some level of persistent programming can be achieved have been developed.

C++ attempts to provide a system in which an object oriented program paradigm is provided above the C programming language [Kernighan and Ritchie 1978]. As such the system attempting to provide a persistent programming paradigm suffer from some unavoidable obstacles. These can be summarised as follows:

- lack of type security,
- lack of pointer control, and
- inability to make some data persistent.

C++ is based upon C, and at any time it is legal for a program to incorporate data structure access with the same loose control as the C language. It is thus impossible for the language to guarantee the use to which any data will be put. Most importantly it is not possible to prevent programs from manufacturing pointers or arbitrarily modifying data. Avoiding such practices can only be managed as a matter of programmer discipline.

The majority of C++ based persistent systems distinguish between data resident within the persistent store and data which is local to the program execution. Local data is ephemeral and is lost upon program exit. It is thus critical that pointers within the persistent data do not refer to objects within the ephemeral area. Such a distinction is counter to the goals of orthogonal persistence. Implementations typically raise a run-time error when such references are made [Campbell, Johnston et al. 1987] or attempt to impose programmer disciplines to avoid the creation of code that creates such references. The ODE (Object Database and Environment) system [Agrawal and Gehani, 1989] is another example of the pitfalls in such a design. ODE provides for persistent objects to be created and deallocated from a persistent heap, whilst ephemeral objects are allocated from a seperate heap. A special class of pointer (a *dual* pointer) is able to reference either persistent or ephemeral objects as needed. However the language allows such pointers to be placed inside both persisent and ephemeral objects, thus potentially allowing a program to create a persistent pointer to an ephemeral object. Only at run time are such an errors detected. The language claims to provide persistence as an othogonal attribute, and this may be true from from the point of view of static language semantics, however when the language cannot prevent a legal program from incurring a runtime error due the nature of the peristence model, the true othogoanllity of the model must be questioned.

Persistent C++ systems generally do not provide the ability to make all forms of program data persistent. In particular they are unable to make process state and program

code persistent. This greatly limits the utility of these systems and again violates the goals of orthogonal persistence.

3.2. Embedded persistence

Another mechanism by which a persistent system can be created is to simply embed the environment into an address space which is itself persistent. Thus, in a manner independent of programming language, anything resident within the address space is persistent. This model is essentially that pioneered by the one-level store of the Atlas architecture.

Such designs are a powerful, they are capable of endowing orthogonal persistence onto any programming language (even assembly languages). However they limit the size of the persistent space to that supported the host architecture.

3.3. Reachability

Many of the languages used in persistent systems provide for automatic storage allocation and control. Automatic storage allocation and recovery naturally leads to a further model of persistence. Recall that persistence of data is only required for as long as it is useful. Clearly unreachable data, data that cannot be addressed from within the programming language, is not useful and need not persist. Persistence of data can be defined in terms of reachability from some distinguished *root of persistence*. By arranging to have a data object reachable in some manner from the root of persistence the programmer can ensure that the data both persists and is useable by other programs that navigate the store. Thus persistence becomes a natural extension of programming in a large object space. Tools for navigating and structuring this space naturally form part of the store itself and a rich programming environment naturally evolves [Dearle 1988; Kirby, Connor et al. 1992; Farkas 1994].

Persistence by reachability can also be implemented within an embedded persistent system. This approach is used by Thatte in the TI explorer [Thatte 1986] and in the Napier88 implementations using the Casper system described in this thesis. These implementations decouple the action of language level garbage collection from the action of the persistent storage mechanisms. The opposite approach has been pursued by Brown [Brown 1994] in which the action of language level garbage collection is deeply enmeshed within the storage mechanism, providing oportunities for optimisations to both.

4. Distribution

The notion of distributed computation is very closely allied with that of concurrent programming. Many of the same issues of protection and control must be addressed, often with the same programming models.

There appear to be three basic models:

1. single unstructured address space,
2. single partitioned address space, and
3. fully partitioned address spaces.

4.1. Single Unstructured

Initial implementations of persistent systems have mostly provided a single address space. The construction of very large stores using this technique was not feasible on conventional architectures until recently due to address size limitations. However, the move toward machines that support such 64 bit address spaces (such as the DEC Alpha [Sites 1993] the MIPS R4000 [Kane and Heinrich 1992] and SuperSparc) has invigorated interest in this approach. The Angel [Wilkinson, Striemerling et al. 1992] and Opal [Jeffrey Chase 1993] systems have adopted this approach in the design of new operating systems. However, there are some difficulties.

i. Establishing a consistent state on stable storage such as disk can become more complex or costly. A single massive address space requires the stabilisation mechanism to either capture the entire state of this space at each checkpoint, or track interdependencies between processes and data in the store and checkpoint interdependent entities together.

ii. If a single address space is shared by all processes, the ability to protect separate areas of the address space must be provided. Page protection mechanisms[Jeffrey Chase 1993] or specialised hardware systems [Eric Koldinger 1991] can be used to provide some protection.

iii. Allocation of free space, garbage collection, unique naming of new objects and the construction of appropriate navigation tools are all difficult to scale, and become unwieldy as systems grow. Attacking these problems has guided the design of the Mneme system described by Moss [Moss 1989].

Distributed Shared Memory (DSM) [Li and Hudak 1989] has attracted considerably interest as a mechanism in which single address spaces can be shared in a distributed environment. However the above problems become more acute as the costs of communication between users of the space become large. Control of these issues leads naturally to some form of internal segmentation structure.

4.2. Structured single address space

In the second model the notion of a single address space in which all objects reside is retained. However, this address space is partitioned into semi-independent regions. Each of these regions contains a logically related set of data and the model is optimised on the assumption that there will be few inter-region references. Providing that control can be retained over the inter-region references it is possible to garbage collect and checkpoint regions (or at least limited sets of regions) independently. The use of such partitioning schemes was introduced by the ORSLA system [Bishop 1977] in the design of an unimplemented hardware system. ORSLA did not identify persistence as an orthogonal attribute explicitly, but did cover many of the requirements for orthogonallly persistent object systems. ORSLA also examined the role of page-based virtual memory in the provision of large addresss space object systems. Partitioning is examined in the provision of distributed access to a single Napier88 store in chapter 6. Techniques for tracking and controlling inter-region references are discussed in Chapter 2.

Different forms of partitioned spaces are used for PS-algol [Atkinson, Chisholm et al. 1981] and the Monads architecture [Rosenberg, Keedy et al. 1992], which uses specialised hardware to control inter-region access and references. The Monads system utilises a form of DSM [Henskens, Rosenberg et al. 1991] in the provision of distributed access.

4.3. Fully partitioned

In the third model the store is fully partitioned. Each region provides a complete and independent address space; there is no global address space. At any time an individual computation executes within a single region and can only access the data visible within that region. There are many advantages to such a scheme. It permits logical grouping of related data, this may improve performance thorough better structuring of disk access and optimised garbage collection. Furthermore, partitioning may be used to provide structured protection between different parts of the system. This can be especially valuable in multilingual environments where type security must be enforced. A structured access

scheme can allow type compatible regions to communicate freely, whilst quarantining incompatible languages. Disadvantages are apparent because of the higher communication costs between the now partitioned processes. In particular, direct addressing of logically shared structures is no longer possible.

4.3.1. Federated

A special case of fully partitioned spaces are *federated* spaces. These are groups of related regions which support the same type systems and are thus able to communicate with their peers without compromising these type systems. If such a federated system supports a suitably rich type system issues of protection can be controlled at the language level. Communication between peers can occur at a high level; such communication paradigms including Remote Procedure Call [Birrell and Nelson 1984] and remote execution [Stamos and Gifford 1990; Dearle, Rosenberg et al. 1991] in which elements of program code are sent from one peer to another for local execution.

4.4. Difficulties

Distributed systems suffer from a number of peculiar maladies that do not affect simple concurrent implementations. These are:

- conflicts in perception due to the inherent asynchrony of communication and execution.
- Partial loss of the execution environment due to either loss of single execution nodes or partitioning of the communication network.

Without extra care these failures will manifest themselves at the user program level. Such manifestation will make visible the location of either processes or data and must compromise the orthogonality of the persistence model. We would argue that one of the goals in manufacturing a distributed orthogonally persistent programming environment is to abstract over the locality of data with respect to distribution in the same manner as locality of storage mechanism.

Replication of data and computation can provide a useful bulwark against some node and network failures, however naive schemes result in loss of referential integrity, which is also fatal to the programming model. To retain referential integrity the system must be able to reconcile replicated data in a manner that is both transparent to the programmer and that preserves the history of the data. Mechanisms which fully capture the causal history of

data in a distributed system are available to do this. In Chapter 7 we examine vector time which is one such mechanism.

A system which tracks the causal history of data can use this information to effect a variety of distributed recovery mechanisms. These designs may make use of replay logging and allow for asynchronous capture of system state on stable media. Such mechanisms may therefore provide avenues for improved system performance. These thoughts provide much of the justification for the discussion in Chapters 6 and 7 of this thesis.

5. Implementation Mechanisms

When providing a persistent environment the designers are faced with a choice of three options. To implement the system using an existing operating system and its host hardware, to design and build custom hardware, or to take an existing conventional hardware architecture and build atop that, exploiting whatever features they can.

5.1. Specialised Hardware

Custom hardware to support an orthogonally persistent system is appealing. The designer is free to create the most appropriate abstractions at the lowest implementation level. The system provides the user with the desired model from the outset without any need to emulate or synthesise using inappropriate abstractions. However a typical custom hardware implementation may have a design cycle of five years, leaving the researchers with a system which is dramatically out-performed by even cheap conventional machines at the end of the cycle.

5.2. Software

Software implementations have been the mainstay of research into persistent systems. They have the advantage that they are very flexible, they can be ported to a wide range of hardware platforms with little effort and that they can take advantage of improvements in the speed of hardware with no effort on the part of their creators. However software models suffer to some extent from inefficiencies in their implementation. This is somewhat unavoidable since they seek to provide a new virtual machine atop a pre-existing virtual machine (the host operating system).

5.3. Conventional Hardware

Constant development driven by a large market and high levels of commercial-competition have delivered and continue to deliver performance gains of nearly a doubling of computation power each year. Conventional hardware platforms are commonly available through out the world; it is easy for other workers to reproduce, test and use the results of research directed at such hardware.

Pure software systems do not, usually by choice, make use of specific hardware features of their host operating environment. To do so would compromise their portability. However some realisations of software architectures do use some of the more common and benign features to improve performance. The two most common features exploited are the ability to directly map pages from files into the address space of a process, and the ability to protect areas of virtual memory from access and then to subsequently handle access exception violations in user code. These features are typical operating system manifestations of the underlying demand paged virtual memory architecture supported by modern architectures. A major thread of this thesis is an examination of the methods by which page based virtual memory can be exploited in the provision of distributed persistent environments.

5.3.1. Granularity

The architecture of page based systems is not ideal for the support of persistent systems. The implementations typically lack flexibility, the grain size of data handled (the page) is often inappropriate, and the efficiency of the mechanisms may be called into question when compared to some optimised language specific software systems. This thesis attempts to characterise the advantages and disadvantages of this approach and to yield mechanisms that make best use of the paradigm. The inherent non-specificity to language is a particular advantage of many of the page based designs presented.

Conventional architectures manage memory in terms of pages, not only for in-memory addressing of virtual memory but also in storage of data on stable media. Strength lies in the simplicity of management of data storage. A page will fit where any other page fits. Allocation of management of memory in fixed size pieces is vastly simpler and more tractable than management of variable sized pieces. Hardware control is simple and fast and operating support is similarly simple and fast.

Operating system support for variable sized data items in addressable memory is possible, the Apple Macintosh [Apple Computer. 1986] is one example. However such schemes require that the operating system is able to compact memory. To ensure that compaction is possible the Macintosh operating system requires that all data under the control of the operating system is addressed through indirection blocks (termed *handles*.) Owners of data are assured that handles are updated to reflect the new location of the data when data is relocated.

Any design for a mass storage device supporting variable length data will suffer from the same limitations, however due to the much larger amounts of data stored and the much slower access times the problems of compaction and allocation become even less tractable. In general systems in which data is managed in fixed sized units can be expected to be implemented simply and perform better than variable sized.

5.3.2. Native code support

Many software architectures result in a system in which user level code does not directly execute the native machine code of the host system, but rather is interpreted by a virtual machine. As implementations advance it is often desirable to exploit the performance advantages of generating native code. Doing so poses some difficulties since the native machine architecture is usually unable to provide the appropriate abstractions of orthogonally persistent memory and the native code is cut off from the support environment available in a software architecture. This thesis will often return to the issues of supporting native code in a way in which minimal impact is made upon the structure of the code, allowing the fullest advantage to be taken of the possible performance benefits.

6. The Thesis

This thesis explores the implementation of persistent systems upon conventional page based hardware. In particular it seeks to explore how a persistent system may be distributed across a group of separate machines without compromising the goals of orthogonal persistence, and the manner in which the attributes of conventional hardware designs can be exploited to best effect in such systems.

In Chapter 2 we explore the manner in which language level mechanisms are implemented using dedicated hardware, software virtual machines, and finally by exploiting page based virtual memory paradigms. The issues examined include layout of data in objects, how to find pointers within objects or pages, the manner in which inter-

object pointers are found and managed, data movement to and from stable memory, and the addressing mechanisms used. Chapter 3 completes the examination of language implementations by examining the manner in which swizzling techniques can be used to increase the size of the persistent space to greater then that supported by the host architecture.

Chapters 4 and 5 examine the mechanisms by which persistent stores are implemented, with a particular emphasis upon page based stores. In particular the implementation of the stores used in the various Casper systems is examined in detail.

Finally we examine aspects of distribution of persistent systems. Chapter 6 explores the implementation of the distributed Casper system. This design allows separate client nodes simultaneous access to a single page based persistent store. It utilises many of the tactics for control of page based systems described in Chapter 2, and introduces the notion of tracking causal interdependencies between separate nodes to optimise the implementation of generation of resilient copies of the system state. Chapter 7 continues the notion of tracking causal links and examines the manner in which a new operating system design (Grasshopper) incorporates causal tracking into its design. Grasshopper is intended to provide orthogonal persistence as an intrinsic attribute and the manner in which the interface between the presentation of persistent address spaces and its implementation by user supplied management code is examined in detail.

6.1. Contributions.

This thesis investigates the manner in which page based virtual memory mechanisms can be utilised in the provision of an orthogonally persistent programming environment, and further investigates the manner in which distribution of such systems may be effected. As detailed earlier, the work has formed part of a number of projects and has received contributions from many people. The particular contributions of this thesis to this body of work are in a number of areas: These are as follows.

In Chapter 2 we introduce the Casper system, the particular contributions to this project included the original concept and design. Also individual design and implementation credit is claimed for the pointer quarantine mechanisms, the design of the extended crossing map mechanisms and its integration into the stable store description tables.

In chapter 3 the issues of pointer swizzling is discussed. As described the origin of pointer swizzling is uncertain, and was probably independently designed by a number of workers at similar times. The invention of pointer swizzling at page fault time is due to

Paul Wilson, the work described in Chapter 3 in extending pointer swizzling mechanisms is an original contribution, as is the taxonomy used for comparison. The use of unaligned access faults to trigger exceptions has been independently observed by a number of workers.

Chapter 4 describes a number of implementation tactics for the provision of reliable and recoverable stable stores. It introduces what we hope is a useful description mechanism and taxonomy which builds upon and unifies a number of earlier taxonomies. Chapter 5 introduces the Casper page based stores, and an implementation of the Napier88 language over this system. The different store architectures and their implementation within both the Mach operating system and under Unix are part of the authors contributions to the Casper project. Chapter 6 describes the distribution of the Casper system, the notion of associations and the use of causal tracking combined with the action of the page coherency mechanism to improve performance of the stability mechanisms is a further contribution. Other contributions are the use of the page coherency mechanism to block execution by denying access to pages when atomic access is required. The author was responsible for the original cache coherency concepts and design in the Casper system. The final version of the coherency system owes much of its success to the efforts of the other members of the project.

Chapter 7 describes the Grasshopper project. This project was conceived and designed by an enthusiastic project team and owes much to many. The particular contributions made by the author are the design of the initial exception handling mechanism and the use of causality tracking mechanisms as an intrinsic part of the operating system design. It is our belief that a persistent operating system, by its very nature, must have an understanding of causal relationships built in from the outset. Much of the experience gained from the Casper project has influenced the design of virtual memory handling mechanisms of Grasshopper although the actual design of these sub-systems must be credited to many project members.

Chapter 2. Implementation Tactics

2.1. Introduction

This chapter introduces the mechanisms by which an orthogonally persistent environment is created. The implementation of such an environment can be achieved by pursuing a wide spectrum of implementation mechanisms, for the purposes of this discussion these are split into the following three categories.

- Dedicated specialised hardware, wherein the processor and support hardware are specifically designed to provide appropriate additional functionality, possibly to the extent of requiring no additional software support.
- A software virtual machine, whereby a virtual machine providing the appropriate abstractions is simulated by a software interpreter.
- Exploitation of existing conventional hardware, in which the functionality of a conventional hardware architecture is exploited to provide some of the functionality provided by specialised hardware, the remainder being supplied by software.

2.2. Issues

A persistent system provides an environment in which the locality of data is abstracted over, thus one of the most important mechanisms we will examine is the movement of data between volatile and persistent storage. In manufacturing a persistent system a further important task is the management of inter data-item references, or pointers. We will devote considerable time to examining the mechanisms by which pointers are recognised by persistent systems. Also examined are how the generation of inter data-item pointers can

be controlled to facilitate storage reclamation and distribution of objects between separate machines. In the next chapter we will examine the problems of translating pointer representations between the forms they take in persistent storage and the forms needed whilst resident within addressable memory.

2.2.1. Data Movement

One important aspect of the abstraction over data locality is transparent movement between different storage mechanisms. In particular the movement between stable storage, where in conventional machine architectures it is not addressable, to addressable memory where it can be used and mutated by a program.

2.2.1.1. Snapshots, Boot and Resume, and Undump.

In the history of object oriented systems such as Smalltalk and the various Lisp Machines lies one of the earliest instances of a form of persistence. Smalltalk implementations on the Xerox Alto architecture [Ingalls 1983] incorporated the facility to take a *snapshot* of the running state of the system onto a demountable disk pack. During normal running the system would periodically create snapshots. A programmer could remove the disk pack and physically take their current environment with them when they moved. Upon re-insertion of the disk pack the Smalltalk system would *resume* the environment at the point of the last snapshot. This feature had some disadvantages however. Writing in [Ingalls 1983] Daniel Ingalls notes "it was necessary to act quickly when fatal errors were recognised, lest they be enshrined forever in the mausoleum of a snapshot. In such circumstances, the alert user would quickly reach around to the rear of the keyboard and press the 'boot' button." Such behaviour became known as *boot and resume*.

Lisp machine environments also provided a mechanism to create an on disk image of system memory from which the system could be reloaded. However it was not intended to act as an everyday mechanism such as the Smalltalk system above. Rather it enabled the large and complex base system to be created which would then provide a fast start up when systems rebooted. Sattish Thatte [Thatte 1986] notes that a system image took of the order of 20 minutes to dump onto disk and this made it unsuitable for general use.

This method (colloquially known as *undumping*) for creating an on-disk copy of complex and otherwise costly to reproduce initial program state continues to be used in programs such as the Emacs text editor and the TeX document preparation system.

2.2.1.2. Data Movement in Persistent Systems

Undumping and boot-resume systems only present a mechanism by which programs can be effectively suspended and then later restarted in a manner which allows the saved state to be preserved across operating system or power failures. Persistent programming is able to offer this functionality, but has far greater scope and flexibility. The ability to share data between programs and the ability of data to be useful beyond the lifetime of the creator programs are some of the additional attributes that are addressed.

This thesis is mostly concerned with tactics which enable page based hardware systems to be exploited to make this data movement as transparent as possible. In addition, this chapter explores the use of specialised hardware support and also explores pure software based solutions.

2.2.2. Address Generation

To access an object, some address must be presented by which the object is known. Thus all useful objects require an address. When a new object is created it is necessary to provide it with an identifier by which other objects in the system refer to it. Also, objects resident within stable storage must be found through the presentation of some address. Addresses within memory and within the stores need not be the same and many variations exist for the management of addresses.

The three main techniques seen in the systems reviewed in this chapter are:

- coincident in-memory and in-store addresses where the address is some form of object identifier or ordinal used to locate objects in the store,
- separate in-memory address and in-store addresses, and
- coincident in-memory and in-store addresses in which the object's location in virtual memory is used for both.

In schemes with separate address representations provision of an identifier is effected in a number of steps. Initially only an address in addressable memory is provided and a persistent address created only if and when the object is moved to stable storage. These schemes perform well because most newly created objects do not survive for very long and are reclaimed before any need to move them to stable storage occurs.

Simple page based persistent systems make addresses for objects within the store coincident with virtual addresses. Such systems do not need to provide separate store

addresses. They are limited to supporting only as many objects as will fit within the host machine's virtual address space.

Page based systems which implement separate address schemes and provide automatic address translation at page fault time [Wilson 1991; Vaughan and Dearle 1992] are also able to extend the address range of objects. This mechanism is considered in detail in the next chapter.

2.2.3. Pointer Representation.

An important aspect of object based system design is the mechanism by which pointers can be identified within objects. In conventional object systems this is of most importance for garbage collection. The mechanisms used in persistent systems need to also take into account the creation of long lived persistent identifiers that may have a different representation to in-memory pointers and may also need to allow the mechanisms that move data to and from stable storage to identify pointers in objects. The method used is usually determined by the languages supported, but may be divided into the following categories:

- tagged pointers supported by software,
- tagged pointers supported by hardware,
- single self describing object format,
- multiple statically known object formats, and
- free form objects with compiler generated format maps.

In addition to simply locating pointers, many language implementations also require the ability to determine the type of each data object. Object oriented languages often need to dynamically determine the type of an object. This may be needed to observe the supported languages type rules, or may be required in object oriented languages to determine the appropriate method instance to apply.

2.2.4. Garbage Collection

Almost all the persistent language systems considered in this thesis utilise automatic storage management. Objects are explicitly created as needed but no explicit deletion mechanism is provided, nor needed. Instead, a separate garbage collection sub-system is responsible for reclaiming space. Whilst programs are running the addressable memory space will require garbage collection. Also the persistent store will require occasional

garbage collection. This is very analogous to generational garbage collection discussed below. It is often possible to continue computation for long periods without requiring recovery of space in the persistent store, only requiring reclamation of the addressable memory space; many systems implement garbage collection of the persistent store as an off-line activity.

A serious problem with garbage collection in stable stores is the need to maintain a recoverable stable state. Compacting garbage collection often involves the modification of almost the entire state of the store and may require almost the same amount of stable storage to hold the modified state as the store originally held. These problems become all the more acute as stores increase in size. The nature of persistent systems is that they are long lived and will continue to grow in size. Store level garbage collection remains an important issue, but is not considered in detail in this thesis.

2.2.4.1. Generation GC

Generational garbage collection [Appel 1989, Leiberman and Hewitt 1983] depends for its success upon the observation that the majority of objects created do not remain reachable (live) for long after their creation, and that the longer an object lives the higher the probability is that it will continue to live. Further it is noted that older objects contain few references to new objects, the converse being the dominant case. Generational schemes attempt to exploit this distribution of lifetimes and references to implement a garbage collection system that can collect the majority of garbage with very short, but frequent garbage collection passes. The key to reducing the execution time of the garbage collection passes lies in reducing the size of the area to be scanned. This is achieved as follows.

New objects are allocated within a small area of memory (termed a generation). Keeping this area of memory small limits the time needed for space compaction and reduces the time of the mark phase. When an object survives for some length of time it is promoted to another (older) generation and copied to a separate area of memory holding that generation. To achieve fast mark phases the location of all pointers to objects within a generation from any older generations are recorded in a separate data structure (termed the *remembered set.*) Maintaining the remembered set usually requires explicit tests upon each pointer assignment to detect the creation of new references to objects in the younger generation. When a generation is to be collected the mark phase uses the normal roots of reachability into the generation plus the contents of the remembered set to find reachable objects.

If objects are moved during garbage collection the remembered set is used to update the value of the referencing pointers (recall that it holds the location of the referencing pointers.)

2.2.4.2. GC in Persistent Systems

It can be observed that the notion of young and old generations corresponds closely to that of the volatile addressable memory space versus the stable store. Using generational techniques when mediating the movement between volatile and persistent memory is, as we shall observe, almost unavoidable. Generational garbage collection has been very successful and has become essentially standard in most object systems. Many of the optimisations that have been developed for generational garbage collection in conventional systems are applicable to persistent systems.

By their nature persistent systems may require very large amounts of data storage. Large stores are difficult to garbage collect easily and whilst generational schemes allow the local context of executing programs to be easily garbage collected collection of the entire store remains difficult. Partitioned schemes (such as the ORSLA design described below) have been designed to improve garbage collection of such large stores. Work by Yong, Naughton and Yu [Yong, Naughton et al (1994)] has evaluated a number of garbage collection systems in the context of client-server systems and concluded that incremental partitioned schemes show the best performance based upon metrics including scaleability, reclustering capability and locality improvement. Further work by Cook, Wolf and Zorn [Cook, Wolf et al (1994)] has shown that policies that select partitions for local garbage collection based upon selecting those partitions that are referenced by pointers overwritten during computation works very well and indeed is close to optimal. Whilst clearly of importance in large scale persistent systems this thesis does not further consider general partitioning designs but concentrates upon the relationship between generational schemes and the nature of persistent data management and the opportunities for exploiting this relationship.

2.2.5. Basic architectures.

The overriding problem in the design of all of the systems described below is the management of data storage, and the abstraction of data location within that storage. A common thread runs through all of these systems based upon the fundamental nature of the hardware used in current machines. The most important of these is the division between

fast addressable memory, which in current hardware technologies is volatile (stored data will not survive loss of power) and high capacity storage which is persistent. A system providing orthogonal persistence must provide for transparent data movement between these two. The issue of representing data addresses is the most noticeable difference between the schemes we will cover. Systems can either maintain the same representation of addresses within volatile memory as is used within the stable store, or elect to perform some translation of addresses. The former are common with systems based upon virtual memory primitives. However, the majority of systems use different representations for data addresses (and indeed sometimes all data) within these storage systems. The use of different address representations is so common that an initial appraisal of the most common generic architecture, utilising an *object table*, is useful before we begin a more detailed study.

2.2.5.1. Object Tables

When an object is placed into addressable memory it will be accessed at the virtual address to which it is copied. When the storage system does not use virtual addresses for its own internal addressing some translation mechanism must be provided to allow identifiers (hereafter generically termed Persistent Identifiers or PIDs) within the store to be translated to in memory addresses. Almost all of the systems reviewed below use some form of translation table indexed by PID as depicted in Figure 1 below.

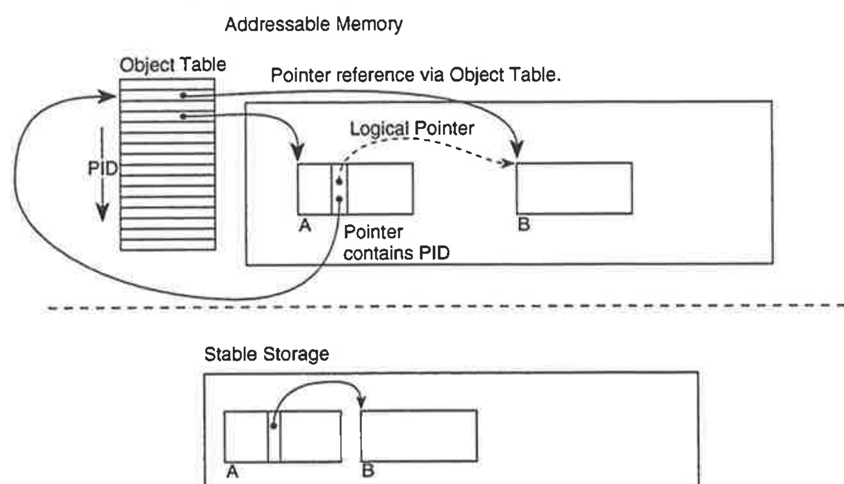


Figure 1 Describing objects in addressable memory using an Object Table.

The use of an object table varies considerably in the various language and support systems to be described. Many systems require all references to objects to be performed via indirection through the object table. This has the advantage of allowing arbitrary rearrangement of objects in addressable memory without requiring a traversal of all objects

resident in memory to update references to moved objects. This simplifies and speeds garbage collection. Such freedom comes at the cost of extra work performed with each pointer reference.

Persistent object management implementations can use the object table to control the location and moment of objects between addressable and stable storage. The table performs much the same function as page tables do in demand paged virtual memory systems, holding the location within addressable memory of objects, modification information, and miscellaneous object attributes. A critically important function of the object table is to allow the system to preserve referential integrity, ensuring that multiple references to an object do not result in multiple copies of the object data appearing in memory. In all architectures, except those which provide a simple persistent virtual address space, with no knowledge of objects, we will find some variation of the object table.

2.3. Hardware

A tempting solution to providing high performance realisations of computer systems is to design hardware that is specifically targeted at a specific problem. We will examine four separate hardware implementations that impact upon persistent object systems. These are the Monads capability based architecture [Rosenberg and Keedy 1987], the Symbolics 3600 Lisp Machine [Symbolics 1984], the ORSLA object oriented architecture [Bishop 1977], and the Rekursiv machine [Beloff, McIntyre et al. 1988; Harland 1988, Pountain 1988]. Of these only the Rekursiv was specifically designed to support orthogonal persistence, however the Symbolics and Monads provide insights into the utility of hardware solutions, and the ORSLA design provides an example of an embeded single address space solution.

In favour of hardware solutions is the ability to explicitly create hardware to support aspects of the virtual machine, often performing the appropriate functions in parallel with other computation. It is possible to create an architecture in which no explicit software support is required to provide the desired abstraction.

Hardware realisations have usually exploited the freedom associated with micro-coded architectures, providing very complex machine instructions as part of the abstract machine architecture. Such designs run counter to the rationale behind the majority of recent architectures which embrace the RISC design philosophy (variously: Reduced Instruction Set Computer, Rationalised Instruction Set Computer) in which an orthogonal but minimal instruction set is provided, instructions which in general execute in a single machine cycle,

and are designed in such a way as to eliminate as many impediments as possible to very fast clock rates and multiple instruction issue. Faced with such competition, micro-coded architectures have faced an increasing performance deficit. Without the investment of considerable effort it is very difficult to increase the speed of these micro-coded architectures, and indeed there is some doubt as to whether it is possible to meet the performance of the RISC architectures at all. As conventional RISC based architectures continue to increase in speed, specialised architectures find that they are out performed by conventional architectures even when the conventional architectures are required to emulate in software the functionality provided by dedicated hardware.

2.3.1. Monads

The Monads architecture encompasses two systems in which very large addresses are supported. The Monads-PC supports a 60 bit address and the Monads-MM supports a 128 bit address. Virtual memory addresses encompas all stable and physical memory. The virtual address space is partitioned into multiple *address spaces*. Each individual address space maintains its own address to disk-address translation table. These tables can be of different formats, as appropriate to the nature of data stored within the address space, however the software in the kernel responsible for data movement must be able to interpret these different formats. This flexibility allows the operating system to implement resilient persistent storage for address spaces that require it.

Monads provides a hardware capability for naming and protection. Programs do not directly address memory relative to address spaces, rather addressing is performed relative to segments which divide the address spaces at arbitrary points. Segments are described and accessed through *capabilities* which, in addition to describing the location of the segment, provide type and access information. Each segment may itself hold capabilities, these may only reside within a segregated area of the segment which can only be modified by a special machine instruction which can only load valid capabilities which the segment has legal access to, thus preventing the forging of capabilities.

To use a segment, a capability referring to the segment must be loaded into one of a number of special capability registers (again a restricted instruction similarly preventing forgery of capabilities). Memory references are of the form:

<capability register> <offset>

The special capability register load instruction will only load values from the segregated capability storage area of a segment, thus the system can create a hierarchical system of references with safety.

The Monads system builds *modules* from groups of segments. A module is used to hold user level objects. The Monads programming model is one in which segments hold programs or passive data, and programs gain access to data through holding capabilities for the segments in which data resides. Capabilities can be regarded as an architecture supported pointer, one which is unforgeable and identified by the hardware. No support is provided for individual programming languages, no structure is imposed nor understood for data resident within segments apart from capabilities. The architecture does not provide special support for language level pointers.

The salient features of the Monads architecture are:

- Hardware support of protected inter-data-item pointers (capabilities referring to segments)
- An integrated virtual memory architecture in which the operating system maintains data on stable media and data is transparently moved from store to addressable volatile memory.

2.3.2. Lisp Machines

The Symbolics 3600 [Symbolics 1984] was designed specifically to support the Lisp language. It features a custom architecture which at the time of introduction provided significant advances in the provision of high performance Lisp environments.

2.3.2.1. Data Format

Every memory word incorporates a set of tag bits that describe the contents of that word. In particular words that contain pointers are differentiated from words containing scalar data. Figure 2 depicts the layout of both scalar and pointer data.

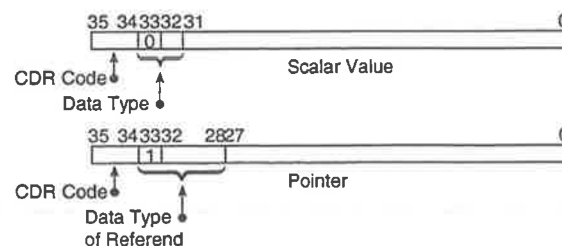


Figure 2. Symbolics 3600 Tagged Word Architecture.

The Symbolics 3600 uses 36 bit words and supports a virtual address space 28 bits wide (256MBytes.) Scalar words are 32 bits in size. Six bits are available for the description of the type of the object referenced by a pointer. Thus the architecture supports a fixed number of internal data types.

A further feature is the use of *CDR-coding*, by which lists of objects can be represented efficiently. Rather than represent lists of simple data types as separate objects, each containing a pointer to the next element of the list, the list is represented as a contiguous array of words, the CDR coding tag bits indicate whether the word is part of a CDR coded list, and if it is, whether it is a internal element or the end of the list. This mechanism effectively represents pointers by the use of an extra two bits in each word.

2.3.2.2. Address Generation and Data Movement

Like the Monads system above, the Symbolics 3600 pages to and from stable storage via a system pager process. However, unlike the Monads system, this does not provide stable storage. This movement is only used for demand paging to increase the apparent size of addressable memory. The pager backing store is not resilient and does not survive between machine bootstraps. Programs which require stable storage of data are required to perform conventional flattening operations to explicitly create a canonical representation on disk and provide code to reverse the flattening upon loading of data. Such schemes must provide their own address generation and translation mechanisms, none is provided nor mandated by the architecture. The system undump mechanism described earlier does however provide a mechanism by which an initial system state can be constructed and demand loaded from store without requiring format translation.

2.3.2.3. Garbage Collection

The Symbolics 3600 offers explicit support for garbage collection. Objects are allocated in virtual memory, and it is only when space within VM is depleted that collection is necessary.

The Symbolics machine implements a form of hardware assisted generation garbage collection. Some areas of virtual memory are designated as *ephemeral*, and garbage collection of those areas is optimised. An ephemeral space can be garbage collected separately from the remainder of the virtual address space, however to do so safely, any pointers outwith the ephemeral space referring to objects within the space must be identified. The Symbolics 3600 does this by maintaining hardware tags for each page of

physical memory. These tags are automatically set when a pointer to within ephemeral space is written to a page. Tags for all of virtual memory are maintained in a special table for use by the garbage collector. When garbage collection of an ephemeral space begins, this tag information is scanned and any page potentially containing references into the ephemeral space is scanned to locate these pointers. Since memory locations are tagged only those words tagged as containing a pointer value need be checked. Garbage collection of the entire virtual address space is still occasionally required. Collection is aided by the architecture's ability to locate pointers in objects, but otherwise runs conventionally. Anecdotal experience suggests that the virtual address space would fill up over a days work and that allowing the system to garbage collect the virtual address space overnight became a common practice.

2.3.3. ORSLA

The ORSLA (Object References in a Single Large Address space) [Bishop 1977] is an unimplemented hardware architecture design.

2.3.3.1. Data Formats

ORSLA provides for a capability based addressing scheme in which the machine hardware utilises tag bits to identify object pointers and prevents the misuse of pointers and also prevents the fabrication of pointers by user level code. Object references are not however simple pointers they also contain other information. The fields of an object reference are as follows:

- The *data_type_info* field (3-5 bits) is intended to convey information such as: whether the object reference contains an address, whether reference counts are being maintained and as yet undefined uses. An interesting use of the *data_type_info* field is in the provision of monitoring of data. If a datum has a special bit in the *data_type_info* field set any attempt to reference that datum will result in the object referenced by the datum's address field being invoked. Thus debugging aids may be implemented and sentinel objects created.
- The *type* field (9-16 bits) is provided into which the type of the referend is encoded. The system provides an initial set of data types and additional types may be added by the user. To avoid depleting the range of data types representable, the designer suggests that those programs that allocate data types incur a charge to be billed to the user ($1000 is mooted).

- The *High/Low* bit controls whether the referend may have load and store operations performed directly upon it (it is a *low level* object) or whether only those operations provided by the object's data type definition (methods) may be executed to mutate it (it is a *high level* object.)
- The *size* field (5-9 bits) allows range checking to be performed on access to objects the object reference also encodes the size of the object. The size encoding is effected using a grainy encoding similar in effect to using a floating point number to represent the size. Thus large objects can only be allocated on coarse boundaries.
- The *address* field (40-50 bits) contains the address in virtual memory of the referend.

The motivation of this complex object reference structure is to allow parallel checking of the referends size, and to avoid the need to store (and read) an object's type in the object. headers. An alternative approach is described in discussion of the Rekursiv design below.

2.3.3.2. Garbage Collection

The particular contribution of the ORSLA design is the introduction of quarantine mechanisms for garbage collection. ORSLA divides the address space into *areas*, which may be individually garbage collected without recourse to traversing objects in other areas. The mechanism predates generational garbage collection but has some strong similarities. In particular ORSLA does not share the strict hierarchy of inter-area reference control provided by generational schemes, but rather treats all areas symmetrically. Generational schemes are only required to track references from older generations to younger and thus allow younger generations to be collected independently. ORSLA allows any area to be individually collected and thus must be able relocate objects in any area whilst preserving the validity of references from any other area . To do this ORSLA uses indirection blocks to mediate all inter-area references. The hardware is required to recognise indirection blocks and automatically chain forward to the actual referend.

To improve performance, pointers between areas may be constructed without indirection blocks. However some mechanism must be provided to ensure that an area which contains references to a second area is garbage collected at whenever the second area is garbage collected. ORSLA uses a construct termed a *cable* to describe such pairs of areas. Cables are directional links and it is the transitive closure of areas referenced via

cables that must be garbage collected when any particular area is garbage collected. The manner in which such directional links can be used to provide a directional linkage of areas provides a hierarchy of object location which is very similar to that provided by generational designs.

The ORSLA system requires that the programmer indicate the area from which a new object should be allocated. The garbage collector may be used to either explicate or automatically move objects from one area to another in an attempt to cluster objects and so improve system performance. It is envisaged that areas provide a similar structuring mechanism to directories on conventional file systems. Users of ORSLA are allocated quotas for storage space and are charged for its use. Areas may be explicitly or automatically garbage collected to reduce the need for persistent storage and may also be explicitly deleted. Explicit deletion of an area may result in dangling references, in ORSLA such references are replaced by a reference to a sentinel object the *deleted* object.

2.3.3.3. Data Movement

The ORSLA system is designed above a relatively conventional demand paged virtual memory system. Movement of data to and from the stable store is effected in the same manner as other more conventional operating systems in response to page faults and pressure on physical memory. It uses associative memory to implement a page table to describe the current state of the system memory. The table is augmented with flags to indicate whether the page is currently being garbage collected (useful when concurrent or parallel garbage collectors are run) and a flag to indicate whether the area to which the page belongs has been deleted. Each page entry also contains the identity of the area to which the page belongs.

Whilst not providing for orthogonal persistence explicitly, ORSLA's use of a single large address space model of programming would allow a trivial implementation of a persistent system if coupled with a shadow page store (such as described in Chapter 5.) The design of the system as presented does not consider the need for such mechanisms.

2.3.4. Rekursiv

The Rekursiv architecture provides intrinsic support for orthogonal persistence. This is achieved through a custom micro-coded architecture. Within the Rekursiv architecture all addressable objects are maintained in physical memory, and access to these objects is mediated by specialised address translation hardware (described later.)

2.3.4.1. Data Movement

When an access to an object that is not resident within addressable memory is attempted, the address translation hardware causes an exception which causes the current instruction to suspend execution. Detection of an object's non-residency is followed by the invocation of a micro-coded routine which mediates the copying of the object from persistent media. This routine is called from within the user level instruction which incurred the original fault. This ability to call micro-coded routines recursively is the origin of the architecture's name. Objects are fetched from disk by a separate sub-system using the object's ID as a key. This sub-system is not part of the Rekursiv and in the only instance of the system realised was implemented in software on a Sun 4/200 series machine which hosted the Rekursiv hardware. This sub-system maintains its own object ID to disk address mappings.

A direct result of the recursive calling mechanism in microcode is the inability to reschedule other processes to take advantage of any idle time available on the processor whilst waiting for the storage sub-system to recover a required object. Indeed the Rekursiv architecture has no notion of multiprocessing.

Once the required object is loaded into addressable memory and the appropriate translation sub-system entries created, the faulting micro-code is allowed to resume execution. Thus the Rekursiv provided an instruction set in which all notion of data locality was abstracted over, and persistence was orthogonal. This is arguably the purest implementation of orthogonal persistence achieved.

Objects within the Rekursiv system are maintained on disk with a description of their size and type. When resident in system memory objects are described by entries in an object table. This table (curiously termed the *pager table*) contains the object's ID, its size, type, location within addressable memory and also the first word of the object data. When placed into system memory, an object's size, type and identity is not kept with it, this information only resides in the pager table. Keeping this data in specialised hardware allows parallel operations to occur during execution. For instance a range check is performed during each object access in parallel with the access itself, utilising the size field in the pager table.

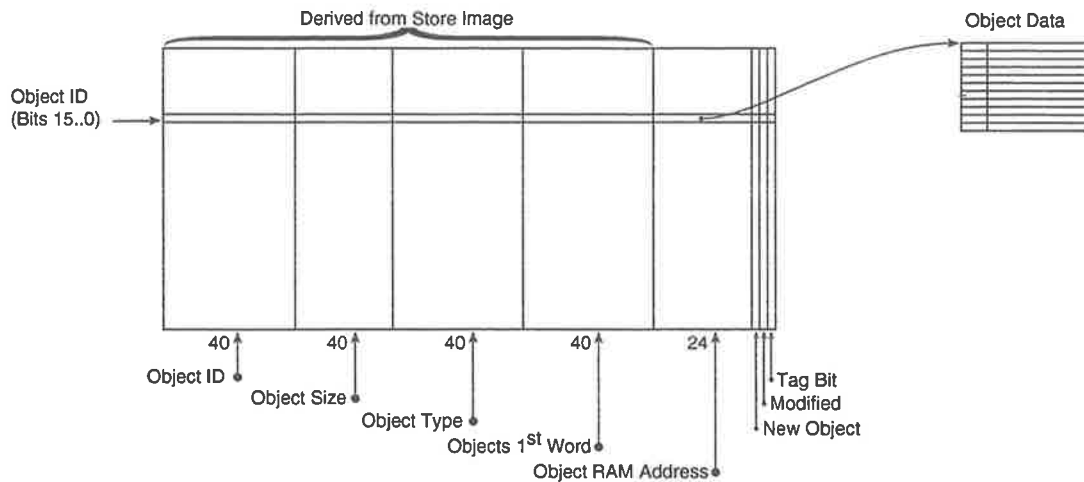


Figure 3 Pager Table format

The pager table is indexed by the object's ID. However the mechanism used is curiously flawed. Being smaller in size (and of fixed size) than the number of object IDs supported, a hash mechanism is needed to fold the object IDs onto table indexes. However the system implemented simply truncates the high order bits yielding an index. No mechanism exists for handling hash collisions, something which drastically compromises the design and complicates the object manager. In particular the object manager must ensure that no objects that clash are ever concurrently resident in addressable memory. This can result in thrashing behaviour if two objects needed for computation clash, since only one is allowed be resident in system memory at once.

2.3.4.2. Object Format

Object representation within the Rekursiv architecture is similar to the that of the Symbolics 3600 in that it uses tagged memory to identify the data type, but differs in one important respect. In a manner similar to the Symbolics 3600 scalar values can be directly encoded within pointers (these are termed *compact objects*). Similarly a small number of types of scalar data are encoded into the tag area. However scalar values that form part of complex objects do not require individual types, since their type is represented as part of the complex objects type.

The Rekursiv also provides some hardware support for complex object types. The type field is held in the pager tables depicted in Figure 3 above. It is not stored in addressable memory and is thus only available to the microcoded support system. Data formats are depicted in Figure 4 below.

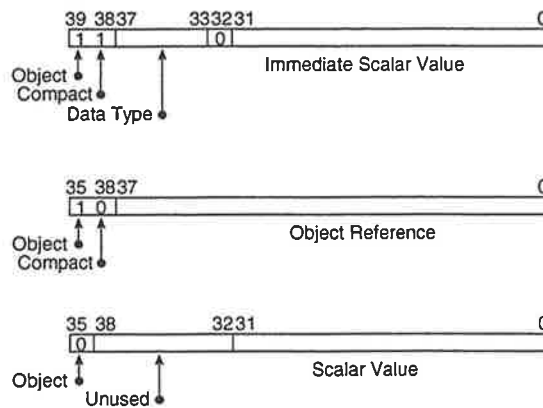


Figure 4. Scalar and pointer values in Rekursiv tagged words.

2.3.4.3. Garbage Collection

The Rekursiv architecture implements a semi-space garbage collection algorithm for reclaiming addressable memory. Objects are allocated contiguously through one half of addressable memory. Once the allocation pointer (a hardware register) reaches the end of the semi-space in use and a request to allocate more space fails, garbage collection is triggered. Garbage collection is accomplished by a micro-coded routine called from within the instruction that attempted to allocate space, this recursive call of microcode occurring in a manner similar to the invocation of the object loading system.

The garbage collection system maintains tag fields in the pager table to mark those objects that may be safely reclaimed. Since all pointers are tagged they are identified with little overhead and no explicit object format is needed. The system must also identify machine registers which hold potential extra roots of reachability, however since the architecture does not confuse data and pointer values, identifying such roots is unambiguous. Once all resident objects are tagged the semi-space is compacted into the unused semi-space. The references to objects in the pager table are updated as compaction proceeds. No explicit support for garbage collection in the stable store is provided, this requires a separate off-line operation.

2.4. Software

The majority of existing persistent systems have taken the approach of implementing a virtual machine above an existing conventional operating system and hardware base. We will examine two software architectures, these are the Persistent Abstract Machine [Brown, Carrick et al. 1988], and the Smalltalk Virtual Machine [Goldberg and Robson 1983]

2.4.1. Smalltalk

The Smalltalk programming language [Goldberg and Robson 1983] and associated programming environment was pioneering and one of the most influential. The original system did not itself embody the ideals of orthogonal persistence, nor indeed persistence as a separate abstraction at all. However, since its introduction considerable effort has been expended in providing support mechanisms for Smalltalk that include varying levels of persistence as an orthogonal abstraction. Further development of storage techniques for object systems in Smalltalk has had considerable influence on techniques used in persistent systems in general. It is a system whose contributions cannot be ignored.

Smalltalk is implemented above a byte code interpreter similar in design to the PAM interpreter discussed next. The Smalltalk environment is provided as a binary image containing the objects and compiled code that represent the environment. The computational model in Smalltalk is one of method invocation in response to the reception of messages by an object. When a message is passed to an instance of a class the message type is used to index a method dictionary to determine the appropriate method to dispatch for the class. Once found the code for the method is invoked and applied to the class instance. In this manner computation proceeds. This mechanism is exploited to aid in object movement in some of the implementations described later.

Beneath the interpreter an object manager is provided. It is with the different object managers and the persistent stores that we are concerned. The original Smalltalk systems were designed when 16 bit architectures were the most common. Accordingly the majority of object systems and implementations were designed around 16 bit representations of integer values and machine pointers. Apart from the LOOM system [Kaehler and Krasner 1983], which is of particular interest because of its efforts to provide a virtual machine supporting 32 bit pointers in a 16 bit environment, we will only concern ourselves with later implementations which were directed at 32 bit architectures, in particular under Mneme [Moss 1990].

The Smalltalk virtual machine has passed through a number of generations, taking it through the Smalltalk -74, -76 and -80 language definitions. In all of these the basic structure has remained the same, an addressable memory space containing objects which are referenced through an object table. The original object manager (named OOZE [Ingalls 1983]) organised the object table as a hash table. This design required computation of the

hash in order to index the table for every object reference. Later versions replaced this with a directly indexed object table, significantly improving performance.

2.4.1.1. Recognising Pointers

In a purely software implementation one mechanism to distinguish pointers from scalar data is to reserve one bit from the word used to represent values. The original Smalltalk implementations [Goldberg and Robson 1983; Kranser 1983] used such an artifice. As described below the differentiation of pointers from scalar values was originally made to optimise representation of scalars, however the effect is of tagging memory to indicate whether a datum is a pointer or not.

In these systems all data is represented by 16 bit words. The system is capable of representing 64k objects with pointers that use 16 bits. However logically all the representable numbers are instances of the integer class (pedantically instances of the class *SmallInteger*.) An integer value should therefore be represented as a pointer to an instance with the appropriate value. Instead, small integers are encoded within the pointer themselves. One bit is stolen from the bits used to represent values and used to distinguish between references to integers and references to other objects. Thus pointers are distinguished from scalar values. SmallIntegers can represent the range -16,384 to +16,383, and the system is able to accommodate 32k other object instances.

Arithmetic is complicated since it is necessary to convert numbers from the SmallInteger representation to the machine representation to perform a computation, and convert back upon assignment. This approach also has the disadvantage of reducing the value space that can be represented.

This software tag scheme has remained in implementations that utilise 32 bit architectures. Figure 5 below depicts the Berkeley Smalltalk implementation [Ballard and Shrirron 1983; Ungar and Patterson 1983] a system designed to provide Smalltalk on the VAX architecture. Figure 5 illustrates an entry in the Object Table referring to an object.

Objects may contain one of either 32 bit tagged pointers, 16 bit or 8 bit scalar data. Only 32 bit values need to be tagged. Therefore objects that do not contain 32 bit values need not use tagged data representations. This is especially valuable for code and for image data. Each object header contains a flag that records whether the object contains pointers, and the object table entry duplicates this flag. The object table also contains a flag that, should the object not contain pointers, indicates whether the object is composed of 16 or 8 bit data.

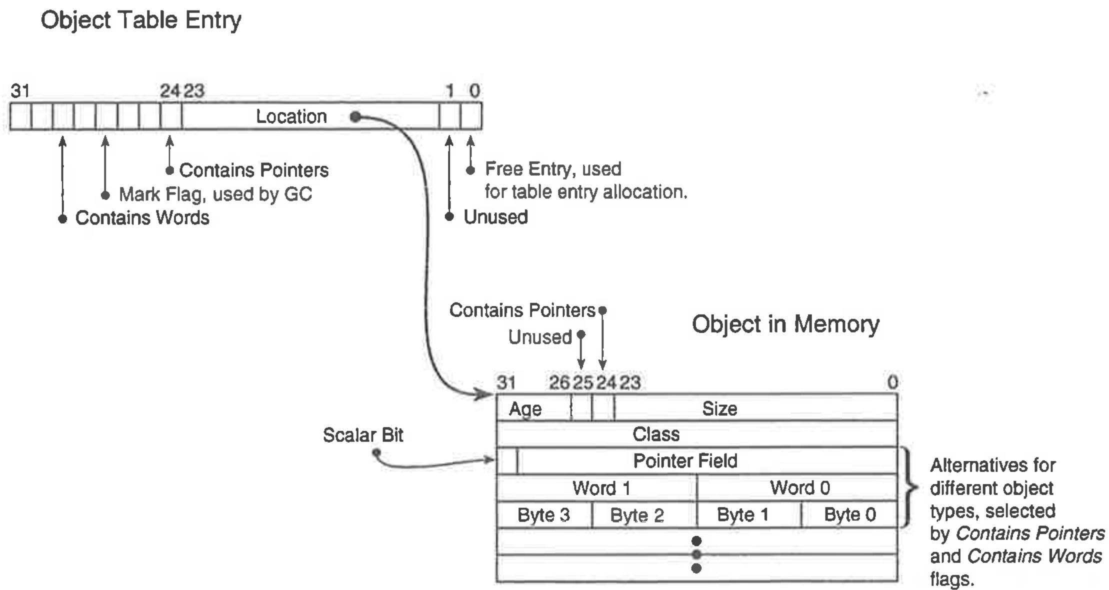


Figure 5. Object format in Berkeley Smalltalk

The Berkeley system utilises a form of generational garbage collection, the object table maintains a mark bit for use by the mark phase, and the objects themselves contain an age field used to select objects for promotion into older generations.

2.4.2. LOOM

The LOOM system [Kaehler and Krasner 1983] is designed to support the Smalltalk Virtual Machine and provide for pointers larger than those representable by the underlying hardware. LOOM is implemented on hardware in which the machine word size is 16 bits. To extend the range of object identifiers LOOM represents identifiers in a stable store using 32 bits. When objects are moved to and from the store the object formats are converted and the identifier fields are overwritten with identifiers of appropriate size.

The basics of the LOOM runtime data structures are similar to those of other Smalltalk implementations and indeed many other object systems. An object table contains the in-memory addresses of resident objects and pointers resident in memory contain an index into the object table. Pointer dereference is performed via the object table. The LOOM system is depicted in Figure 6 below.

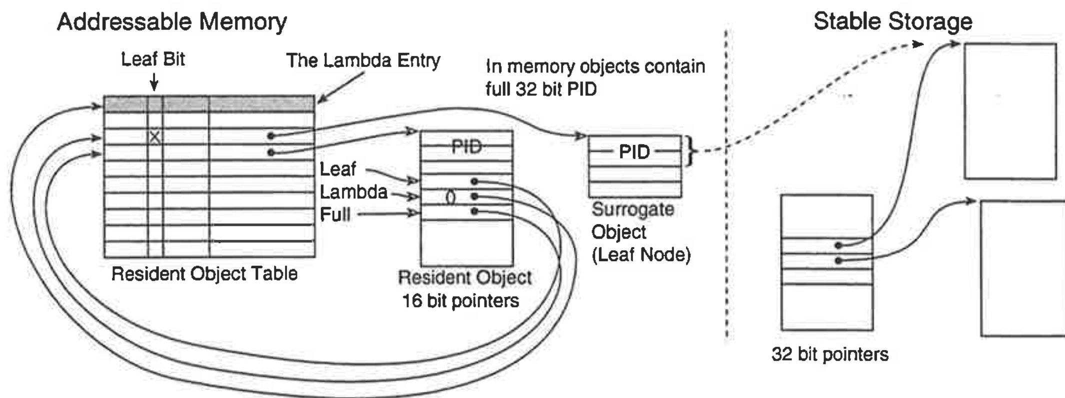


Figure 6. The LOOM system.

When an object is loaded form stable store the objects are translated in form, the 32 bit (or larger) object pointers are replaced by 16 bit indexes into the object table. Each in-memory object is also prefixed by its 32 bit PID. Pointers cannot exist in memory in 32 bit form. Thus in memory pointers to non-resident objects must be specially represented. There are two special forms of in memory pointer provided for this purpose. These are the *leaf* and the *lambda*.

An object can be represented by a leaf node which acts as a surrogate for the real object. The leaf node only contains the real ID (32 bits), size (always 4), and reference count information used by the garbage collection system. The object table incorporates a flag to indicate if an object is represented by a leaf node. The method dispatch mechanism checks the object table entry and activates the object fetching mechanism as required. When a reference occurs, the real object is copied into memory and the object table entry overwritten by a reference to the actual object. The leaf object is discarded. Since all references occur through the object table all the pointers to the object will now be correct.

All leaf nodes require an entry in the object table, whether they eventually are dereferenced or not. To conserve object table space, pointers may be replaced by a distinguished pointer, the *lambda*. This is the value zero which indexes a special entry in the object table. When a reference to a pointer containing the lambda value occurs the system must recover the real value of the pointer from the copy of the object in the store. The object is copied from the store into a buffer and the 32 bit pointer value extracted. The object faulting system then loads the referend object and overwrites the lambda pointer with the appropriate index to the newly loaded object's table entry. Care must be taken with lambda pointers: pointer assignment must ensure that lambda values are never written

into pointers since there is no mechanism for later determining what the intended referend was.

Checking for lambda pointers during execution slows down execution speed. To avoid checking for lambda wherever possible, each resident object is tagged as to whether it contains any lambda pointer entries. The run-time system ensures that those objects in the immediate context of the program do not contain lambda pointers by eagerly replacing entries by leaf or full pointers. Using the lambda pointer mechanism is only a gain when these pointers are unlikely to be referenced. To help choose whether to create lambda or leaf references when an object is loaded the system maintains hints within the images of the objects in the store so that previous history can be used as a guide.

The LOOM system is of special interest as it is one of the first in which pointers are represented in a different form in addressable memory to that in stable store. The overwriting of pointer values with different representations has become known as *swizzling*. The LOOM systems ability to represent a larger virtual address space than that provided by the underlying hardware forms the inspiration of the systems described in the next chapter.

2.4.2.1. Smalltalk and Mneme

Hoskings and Moss [Hosking and Moss 1993] describe an implementation of Smalltalk above the Mneme persistent store. This system is implemented on 32 bit hardware, but has considerable similarities with the LOOM system above. Its general operation is depicted in Figure 7 below.

Objects are copied from the Mneme persistent object store. Mneme transfers data in *physical segments*, which have arbitrary size. Clustering of objects within physical segments causes an entire cluster to be fetched from the store at once, avoiding the performance problems of single object faulting in LOOM. Representation of pointers from resident objects to non-resident objects is achieved through two mechanisms akin to the leaf and lambda of LOOM. A surrogate object may stand in for a non-resident object, these are termed *fault blocks*, and the mechanism termed *node marking*. Fault blocks serve much the same purpose as leaf nodes in LOOM. Since a 32 bit architecture is used and the underlying Mneme store uses 28 bit PIDs only a single word is needed to hold a fault block. This implementation does not perform dereferences through the object table, rather pointers to resident objects contain the VM address of the object. This removes the cost of indexing the object table, however it is no longer possible to simply discard the fault block

since other pointers in memory may still reference it. Therefore when an object referenced by a fault block is copied into memory, the fault block is overwritten with the address in memory of the object. The overwritten fault block is now termed an *indirection block*. The running system must be able to identify indirection blocks and transparently redirect accesses to the real object. This adds a small cost to the running system. Indirection blocks are eliminated when found by the garbage collector and are also eagerly removed at object fault time.

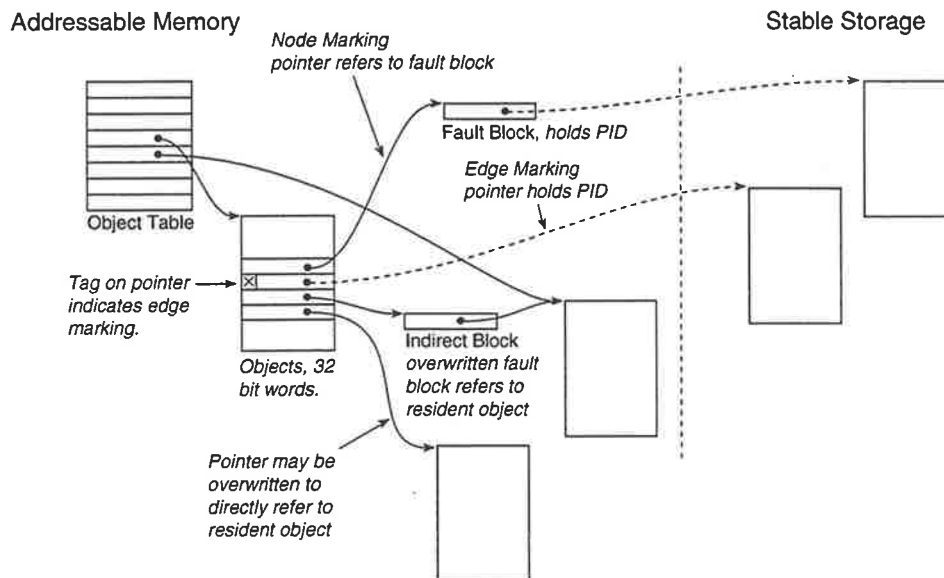


Figure 7. Node and Edge marking to refer to non-resident objects.

Instead of referencing an indirection block, individual pointers may contain the Object Identifier of the referend. This is termed *edge marking*. This field requires a tag to differentiate PIDs from valid pointers, again since PIDs only use 28 of the 32 bits available, there is space for the tag.

It would appear that both edge marking and node marking are equivalent, since both fault blocks and edge marked pointers hold the PID of the referend. The difference comes about through the mechanisms used by the Smalltalk system to activate object fetching.

Objects are brought into memory when a dereference through either a fault block or an edge marked pointer is attempted. There are three mechanisms through which detection of references to non resident objects may be achieved:

- explicit tests upon each dereference,
- folding the detection of such dereferences into the method dispatch mechanism, and

- placing the fault blocks in a protected memory region and using the resultant exception to trigger object fetching.

To understand how method dispatch is used to load objects we need to review the internals of the method dispatch mechanism.

The computational model in Smalltalk is one of method invocation. Messages are passed to an instance of a class and the message type is used to index a method dictionary to determine the appropriate method to dispatch for the class. Determining the appropriate method is complicated by the inheritance structure. To improve performance a method cache is used. This is indexed by method selector and class, and returns a reference to compiled method code. To avoid the need to check whether the method code is resident in addressable memory the system ensures that the class of any object is loaded when the object is loaded

As described earlier, the Smalltalk system uses a tagged architecture to distinguish pointers from scalar values. Scalars are termed *immediate* since their value is always immediately available. Upon method dispatch, the system must test each value to determine whether it is a pointer or an immediate value. Edge marked pointer references are marked as immediate values (although logically pointers they do not directly reference an object) and are considered to be members of the special class *null*. There is only one method in the class *null* and it responds to all messages by first loading the object referenced by the pointer into addressable memory and forwarding the message to it. Thus, in principle, objects are faulted with no run-time check for residency. Arguably the cache lookup system must reflect the special tagging and mapping to the null class; it may therefore be slightly more complex than a straight forward hash function. The additional cost of this extra functionality is low, but probably not zero.

This mechanism can be seen to have distinct parallels with the lambda pointer mechanism used by LOOM. In LOOM the lambda entry explicitly indexes the object load method, edge marking uses the hash function to perform the same action.

When a fault block represents an object a mechanism essentially the same as that used by LOOM is used. Fault blocks respond to all messages by loading the object which they represent. However fault blocks only contain a PID and are not full objects, therefore they do not contain a class pointer. The method dispatch mechanism must therefore test to see if the referenced object is a fault block and explicitly invoke the object loading code. This extra test adds to the normal running costs of the system.

Moss and Hoskings also outline a method by which these tests can be eliminated by utilising memory protection faults. By placing fault blocks within a protected memory region any attempt to access the fault block will lead to an exception which is used to trigger object loading. They present test results [Hosking and Moss 1993] which show little difference in performance between these mechanisms. They conclude that little is gained by using memory protection for their system. The test essentially balances the small extra cost added to each method dispatch against the occasional but larger cost of the exception handling. The issue of exception speed and alternatives is covered in detail in section 5 of this chapter.

2.4.2.2. Garbage Collection

In a conventional generation scheme when an object is promoted to an older generation the younger generations must be scanned to find and update any pointers which refer to the moved object. Hosking [Hosking 1991] describes a generational garbage collection scheme in which multiple *chunks* of objects are maintained in addressable memory. Hoskings scheme avoids the need to update pointers when objects in a chunk are moved by overwriting the moved objects with an indirection block. Thus a moved object will be accessed transparently by a running program. Chunks are a coarse grained allocation structure. New objects are always allocated into chunks, and the system actively reclaims memory chunk by chunk. The system attempts to free all the memory in a chunk in two stages.

First it *closes* a chunk so that no further objects are either created or copied into it. It then proceeds to remove the objects resident within the chunk. Objects are lazily copied either into other chunks, or back to the stable store. To ensure references to moved objects remain valid, indirection blocks replace the moved objects. If the object has been returned to the store, a fault block replaces the object.

Once all incoming pointers to a indirection and fault blocks in a chunk are bypassed the chunk may be freed. The system bypasses references when it can. The generational garbage collection system aids in this. The remembered set contains the identity of all pointers from older generations referring to objects in younger generations, thus the system can find all the pointers into a chunk from older generations without the need to traverse the entire contents of memory. When the garbage collector runs it automatically bypasses references to indirection and fault blocks from younger generations. Thus once a garbage

collection pass has run within the younger generations, no pointers from younger generations to a chunk will remain.

2.4.3. PS-algol and CPOMS

The language PS-algol is the result of original research on the integration of programming languages and data-base management systems [Atkinson 1978]. Persistence as an orthogonal attribute was added to the S-algol language [Morrison 1982]. Implementation of PS-algol has been provided by three systems, namely: the chunk management system [Atkinson, Chisholm et al. 1983], the PS-algol persistent object manager [Atkinson, Bailey et al. 1983], and finally the CPOMS (Persistent Object Manager in C) [Brown and Cockshott 1985; Brown 1988].

2.4.3.1. Computational model

PS-algol makes persistence of data as transparent as possible, in particular:

- the programmer does not have to explicitly identify persistent data,
- all data may be used independently of whether it is persistent,
- all data types may be made persistent, and
- the language's protection mechanisms apply to all data.

The persistent store provided by the CPOMS is logically divided into separate *databases.* Each database provides a root of persistence from which all objects within the database are reachable. Databases may be separately updated using a transactional mechanism. This mechanism allows the database to be used as a unit of sharing between different programs. Before a database can be used it must be locked and the intention to modify data made. Changes are made permanent in the database with an explicit call to the *commit* function.

2.4.3.2. Addresses in PS-algol and CPOMS

Utilisation of many databases complicates the creation of addresses in PS-algol. Although each database is a separate entity, addresses of objects within each database must be distinct to allow programs to use arbitrary combinations of databases at one time. The CPOMS uses a large address space that is divided into relatively small fixed size partitions and individual databases use as many of these partitions as they need.

A persistent store (database) address is divided into two parts: a partition number, and a logical address within the partition. Each database maintains an indirection table which maps the logical database address to the actual location within the database file. This indirection is provided to ease store garbage collection, and is discussed later. Databases

themselves are identified from the partition number through a further translation table the PTODI (Partition TO Database Index).

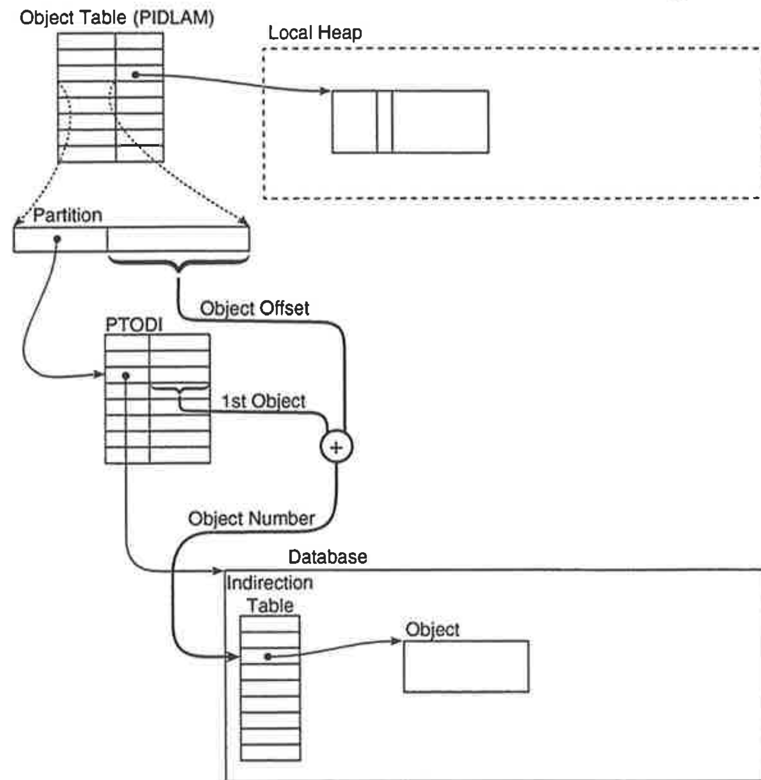


Figure 8. Address translation in CPOMS.

Objects resident in addressable memory contain pointers either in the store format just described or contain the address in addressable memory of the referend. An object table (termed the PIDLAM Persistent IDentifier to Local Address Map) is used to provide a mapping from PIDs to addressable memory locations in the manner previously described for other systems. The interpreter is responsible for checking the nature of the address (with a runtime check performed prior to each dereference) and for initiating translation of the address or fetching of the object from the store as appropriate. The address translation structure is summarised in Figure 8 above.

Newly created objects are allocated PIDs when they are placed within one of the databases. Allocating PIDs is performed as part of the commit algorithm described next.

2.4.3.3. Data Movement

When a PS-algol program starts running the system has no means of directly addressing the persistent store, it only knows the names of the databases. Access to individual databases is achieved by the program *opening* them. A database directory is maintained which maps the name of a database to the appropriate file name and also records a password which can be used to protect access to individual databases. Once a database has

been identified it is locked appropriately and the first PID address partition it contains is placed in the PTODI table. Once this is effected the system can proceed to access objects within the database.

Databases may contain references to other databases, to avoid run-time errors resulting from an attempt to access an object referenced in a database that was not explicitly opened, the system opens the transitive closure of databases reachable from the database which was first opened. Each database maintains a table of databases which must be opened if it is itself opened. These extra databases are opened for read-only access unless they do not have explicit names (in which case it is impossible for a user program to name them and hence open them for write) in which case they are opened for write access.

Objects are copied from their host database on demand when the running system encounters a PID. Object copies are placed in the local heap and an entry made in the PIDLAM. Modified objects are written back to their home database as part of the commit operation. Only objects from databases that were opened for write operations can be written back. Not only must modified objects be written back as part of the commit operation, but so must those newly created objects which are reachable from persistent objects. PIDs for these new objects are allocated so that they will reside within the same database as the persistent object that references them.

However more than one persistent object may reference a new object, and these objects may have different home databases. When this situation arrises an entry is placed into a table in the second parents home database which lists those databases which must be opened if this database is opened. Thus the new object will be accessible no matter which database it is referenced from.

Since databases may be opened for read-only access the system will refuse to commit any objects to any databases if it is found that an object read from a read-only database has been modified.

2.4.3.4. Object Formats and Identifying Pointers

PS-algol does not provide a single canonical object format, rather it defines separate individual object formats for each base type supported by the system. These include: strings, structures, stack frames, code vectors and raster images. Use of separate object formats requires that each of the sub-systems understand each of the formats. In particular:

- the compiler must be able to generate code to use each of the formats and create objects in each format,

- the abstract machine must be able to create and mutate objects of each format,
- the garbage collector (both for in store and addressable memory object representations) must be able to find the size of, and pointers resident within, each format and,
- the store system (CPOMS) must be able translate each object format appropriately on movement to and from the persistent store.

Any change to an object format, or the addition of a new object format, requires changes to each of these sub-systems. This was identified as a significant disadvantage of individual object formats [Brown 1988].

2.4.3.5. Garbage Collection

Like the other systems described in this chapter, PS-algol implementations provide separate garbage collection for data in addressable memory (the local heap) and stable storage (the databases).

Garbage collection of the local heap is performed in a conventional manner except that the PIDLAM also acts as a root of reachability. This ensures that objects which are reachable from objects which reside in a database but not from any object in the local heap are not discarded.

Garbage collection of the databases is performed by a separate program, and runs on otherwise quiescent databases. Two mechanisms are provided for garbage collecting a group of databases.

A fast mechanism is provided which uses the list of referenced databases held within each database to build the closure of all reachable databases. It is possible that there exist unreferenced anonymous databases which can then be reclaimed.

A full garbage collection system is also provided. This traverses the objects held in all of the reachable databses and builds a single list of all reachable objects (by PID). Once complete the list is used to compact each database in turn. A new list of referenced databases is also generated for each database as part of the compaction phase.

2.4.4. Napier-88 and the PAM

The Napier88 language [Morrison, Brown et al. 1989] was built as a platform for further experiments in persistence in languages. Its design is partly derived from experience with shortcomings in the PS-algol language. In Napier-88 all data is persistent. A single root of reachability (returned by the special function `PS`) is provided and any object reachable from

this root remains persistent. When a program instance terminates all data that is not thus reachable is reclaimed.

2.4.4.1. PAM

In the course of this thesis we examine various implementations of the programming language Napier88 in considerable detail. Of particular relevance to the discussion of software architectures is the software architecture that provides the mainstay of existing implementations. This is the *Persistent Abstract Machine* or *PAM.*

The PAM [Brown, Carrick et al. 1988] is a byte-coded interpreter, and was primarily designed to support the Napier88 language. Due to the modularity of its design and implementation, it may be used to support any language with at most the following features: persistence, polymorphism, subtype inheritance, first class procedures, abstract data types, block structure and assignment. This covers most algorithmic, object-oriented and applicative programming languages.

One of the notable features of the PAM is that it is constructed entirely upon a heap-based storage architecture. This heap-based architecture was considered advantageous for the following reasons:

- Only one storage mechanism is required, easing implementation.
- There is only one possible way of exhausting the store. This was considered an essential requirement, in that applications can only run out of store when the persistent store is exhausted, and not merely when one of the individual storage mechanisms is exhausted.
- The PAM is designed to support languages with first-class procedures, block structure, free variables and assignment. A direct implication of this is that L-values may persist after their names are out of scope. This property is known as block retention [Berry 1971] and is not required for most languages.

Stacks are still used conceptually, with each activation record being implemented as an individual heap object. Each of these records represent the frame (activation record) required to implement a block or procedure execution. In the case of Napier88, the size of each record may be calculated statically. A consequence of activation records being allocated from a heap rather than a stack is that the heap is heavily used for allocating store for short lived objects. These short lived objects may be garbage collected provided that transient objects may be distinguished from persistent objects resident in the heap.

Beneath the PAM, a persistent object store is provided. The original implementations of this object store used a layered architecture, each layer providing a specific abstraction, the lowest directly managing stable store, through to the highest which presented an abstraction of persistent objects. We will explore the implementation of this system in some detail. This thesis will also describe a new implementation of a persistent virtual address space which takes advantage of the page based virtual memory and the exception handling mechanisms provided by conventional hardware architectures.

2.4.5. The POMS Store.

The POMS (Persistent Object Management System) provides one implementation of a persistent object system utilised by the PAM for the provision of a Persistent Store.

The POMS provides a procedural interface directly callable by the PAM. As programs are executed by the PAM, it tests for object residency and invokes the POMS to mediate the movement of data from stable storage to addressable memory. Addressable memory used by the PAM is managed as a heap of objects and individual objects can be copied from stable storage to the heap on demand. Since the PAM architecture is interpreted, tests for residency are only performed once per PAM instruction even if the object is accessed many times during an instruction, reducing the number of tests somewhat.

The POMS allows two separate representations of pointers to objects: the object's virtual address in addressable memory and the object identifier (PID). These are distinguished from one another by ensuring that all PIDs are negative numbers, thus a simple sign test serves to determine which kind of pointer is being used. If it is determined that a pointer is a PID, a further test is made to see whether the referend is resident, if not the object is explicitly loaded. Once the object is resident within addressable memory the instruction is restarted and proceeds normally.

2.4.6. Object Format

The POMS uses a single object format. The most important feature of this format is the segregation of pointers from scalar data. No matter what the type of the object stored it is always possible to find all the pointers in an object. Unlike the Smalltalk systems described earlier, pointers are not tagged to differentiate them from scalar values and are not intermixed with scalar data. The POMS object format is depicted in Figure 9 below.

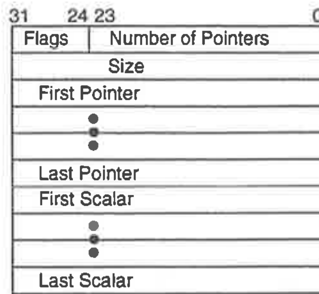


Figure 9. Object format in POMS

Flags are maintained in the object header and include a mark flag used by the garbage collection system, a modification flag used by the persistent store manager and format flags that can be used to indicate different byte addressing schemes for different hardware architectures.

2.4.6.1. Data Movement

The POMS system is a layered architecture in which different strategies can be employed in implementing each layer. The implementation by Dave Munro [Munro 1993] is of interest as it utilises some of the attributes of page based virtual memory that will be the subject of the next section. In this scheme the store is presented to the POMS as a memory mapped file, a strategy which allows objects to be copied directly into the local heap as a simple memory to memory copy, without recourse to the conventional Unix file manipulation primitives. This is illustrated in Figure 10 below.

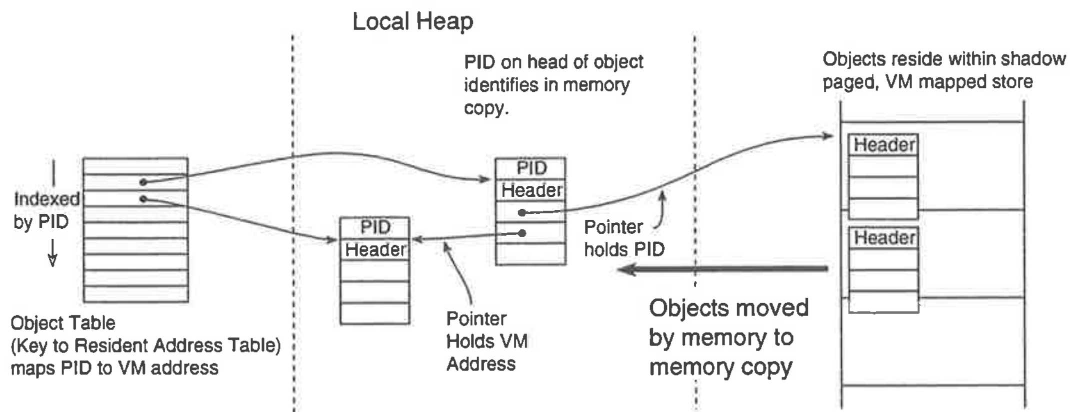


Figure 10. POMS with memory mapped store.

The operation of the memory mapped store is described in Chapter 4. Objects resident within the local heap are prefixed by their real PID, this allows the POMS to retrieve the PID of any object referred to by virtual address pointer, allowing for fast replacement of in memory pointers by PIDs. Should an object not have a PID the prefix contains a sentinel value.

2.4.6.2. Address Generation

As the PAM creates new objects they are created within the local heap (addressable memory) and are referenced through the virtual address at which they reside. The majority of created objects will be reclaimed before any action is taken to write the state of the system back to stable storage. Those objects that survive until either they, or another object which refers to them, are written back require the generation of a PID by which it will be known in the persistent store. This is accomplished as follows. Whenever an object is written back to the Persistent Store it is scanned to find any pointers referencing objects that do not have PIDs. If any are found a PID is allocated and an entry made in the object table. The pointer is overwritten with the PID before the initial object is written to the store. If the object being written back itself has no PID, one is similarly generated and entered into the store data structures. Thus objects within the store only contain references in the form of PIDs. Notice again the similarities to generational garbage collection; the movement of objects to the store closely corresponding to the promotion of objects to an older generation.

2.4.6.3. Garbage Collection

The PAM system garbage collects objects in both the local heap and in the stable store. Garbage collection occurs in a running program by collection of unreferenced objects in the local heap. The local heap is only a cache of objects resident within the store and care must be taken to ensure that all objects reachable from objects resident in the persistent store are retained. In particular when an object is evicted from the local heap and copied to the store all those objects referenced by pointers within that object still resident in the local heap must not be reclaimed even if no references to them exist within the local heap. To ensure that these objects are retained, the system ensures that all such objects are allocated a PID and are marked as modified. The local garbage collector may not reclaim such objects. These objects will migrate to the persistent store upon commit. A list of modifed persistent objects serves a similar purpose as the remembered set in generational schemes. However since objects in the store (the older generation) only refer to objects in the younger generation (the local heap) by PID there is no need to overwrite these pointers with the referends new location. Thus there is no need for the full functionality of the remembered set of conventional generation garbage collection.

The canonical object format of PAM objects allows the garbage collector to find all pointers in objects without recourse to external information.

2.5. Page Based

This thesis is partly concerned with the use of *conventional* page based architectures to support orthogonally persistent systems. The definition of *conventional* is essentially those architectures which comprise the majority of available computer systems. Architectures based around the commonly available hardware implementations such as Sparc, Motorola 68000, 88000 and 600 (PowerPC) series, Intel *x*86, MIPS and DEC Alpha AXP. The features common to almost the entire spectrum of available hardware platforms of interest in providing persistent systems are:

- paged based virtual memory
- exception handling mechanisms.

These two mechanisms are inextricably intertwined in most systems. The provision of demand paged virtual memory being dependant upon the provision of a suitable exception mechanism. Appel and Li surveyed the possible uses to which virtual memory primitives can be put in the support of such systems [Appel and Li 1991]. They identified the following areas:

1. *concurrent garbage collection*, the garbage collectors of Ellis, Li and Appel [Ellis, Li et al. 1988] utilise page protection to provide a barrier between the mutator and garbage collector, allowing them to run concurrently. Page protection prevents the mutator from accessing memory ranges within which the garbage collection process has not completed.
2. *shared virtual memory*, distribution of a shared memory execution model, utilising page protection and access exceptions to trigger the coherency mechanisms.
3. *concurrent checkpointing*, wherein page movement akin to demand paging is used to create snapshots of the address space on stable media concurrently with mutator execution.
4. *generation garbage collection*, utilising page protection and resultant exceptions to detect the creation of pointers between generations.

5. *persistent stores*, in a manner similar to demand paged virtual memory the virtual address space is maintained as a resilient and recoverable store, providing a persistent virtual address space.
6. *extended addressing*, an illusion of an address space considerably larger than that supported by the underlying hardware is provided by translating the format of pointers within data when pages of data are faulted from the persistent store.
7. *data compression paging*, in which pages of data are represented on disk in a compressed form and compressed or uncompressed when the data is moved.
8. *heap overflow detection*, through the use of guard pages to detect attempts to reference past the allocated area.

This thesis will explore the implementation of all but the first and seventh issues on this list in some detail.

2.5.1. Data Movement

Persistent implementations that take advantage of page based virtual memory are able to exploit the ability of the system to transparently provide residency checks for pages of data, and to provide transparent fetching of the required data. This provides some considerable advantages:

- user code need not contain any checks for data residency,
- user code need not contain any code to mediate data movement,
- user code is therefore able to execute faster when data is resident,
- user code is smaller, further improving execution speed,
- creation of persistent environments for primitive languages (such as C, and assembly languages), and
- compilers do not need to provide any support for the underlying persistent system. It is therefore possible to provide an orthogonally persistent system using any language and compiler.

This freedom is bought with two costs:

- The granularity of data and data movement is the page and is fixed by the choice of hardware.

- The cost of trapping the exception raised by non-residency can be high on some architectures [Appel and Li 1991; Hosking and Moss 1993; Chandramohan and Levy 1994].

2.5.1.1. File Mapping

File mapping is a natural extension of demand paging mechanisms. Indeed the VMS [Levy and Lipman 1982] and Mach [Acceta, Baron et al. 1986] pager systems use files within the user file space to perform process paging and swapping. Almost no change is required to these operating systems implementations to allow users to specify an arbitrary file from which to perform paging for a section of a virtual address space. Recent versions of Unix have also offered this functionality. Once a file is mapped into a process's virtual address space, data within the file becomes visible and directly addressable. Data movement is mediated by the operating system's normal page fault mechanism and occurs transparently to the execution of user level programs. Data modified by a program's execution will be returned to the mapped file as a direct consequence of the action of the paging mechanism, and is automatically written back upon closure of the file. Thus data resident within the address range served by the mapped file becomes persistent.

However simple files maps are unable to provide a resilient persistent store. In particular, no control is afforded over when data is written to the store file, and at any time the contents of the file on disk may contain pages of data that individually represent the execution of the system at widely varying times. Thus it is very unlikely that the store file contains a useable representation of the executing system. Should the system suffer a failure during use of such a store the system would probably not be recoverable. Mechanisms to manufacture appropriate resilient mechanisms using shadow paging mechanisms are explored in Chapter 4.

2.5.2. Texas and ObjectStore

Newer implementations of persistent systems also use the memory map mechanism provided though conventional operating systems. Both the Texas [Singhal, Sheetal et al. 1992] and ObjectStore [Lamb, Landis et al. 1991; Object Design 1994] system utilise the file map capabilities provided by the Unix [Ritchie and Thompson 1978] operating system.

Both the Texas and ObjectStore systems provide an addressing environment larger than would ordinarily be possible within the limitations of the host architecture. This is accomplished by allowing PIDs to be larger than virtual addresses and overwriting them

with virtual addresses when objects are copied into virtual memory. A full exploration of this mechanism is provided in chapter 3.

2.5.2.1. ObjectStore

ObjectStore provides a persistent environment for the C++ and Smalltalk programming languages. Of particular interest is the manner in which ObjectStore uses the host operating systems virtual memory system to aid in providing access to stored data.

Object store provides a model of computation very akin to that of PS-algol. Users can construct individual databases and an object becomes persistent by residing within a database. Access to databases takes place during transactions. During the period of a transaction the database is provided to a user program by supplying pages containing the appropriate data from a store server and mapping these pages into the user's virtual address space. Once the transaction has completed these pages are unmapped. However to improve performance when a series of transactions may access the same pages, a cache of pages is maintained in the client so that they may be remapped without the need to re-request them from the server.

2.5.2.1.1. Data Movement in ObjectStore

Requests to load individual pages in ObjectStore are mediated by page faults. An attempt to access a pointer that refers to data on a non-resident page causes an access exception which invokes the memory management sub-system. ObjectStore's operation is illustrated in Figure 11 below.

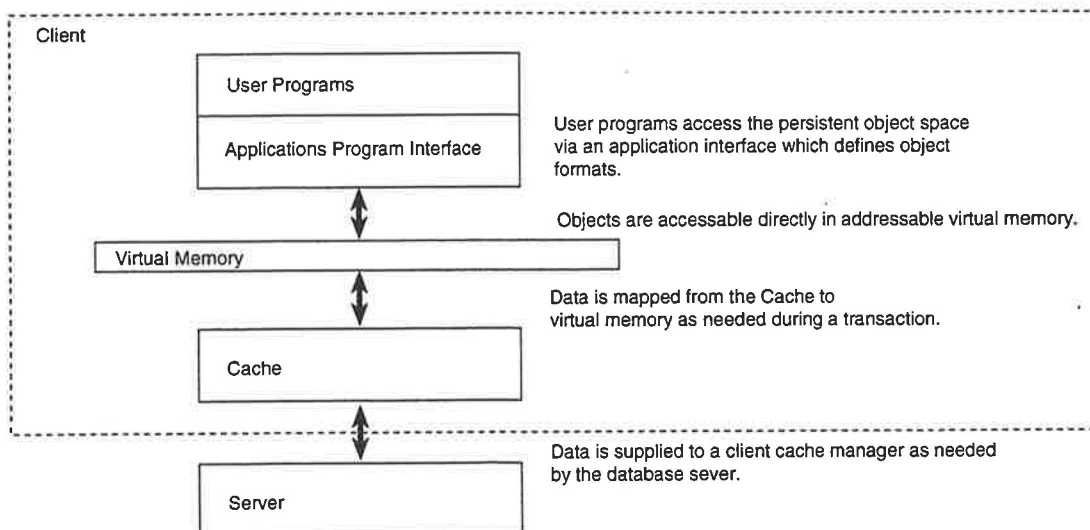


Figure 11. Simplified view of ObjectStore operation.

ObjectStore may potentially map a page of data at any available address. This requires that references to objects on this page be translated to refer to the correct virtual address.

However the system designers claim that if it is possible to map a page at the address at which the in store object identifiers match the addresses in virtual memory, it will do so. The method by which ObjectStore allocates memory and translates pointers seems to very similar to that described by Wilson for the Texas system and discussed in the next chapter. However at the time of writing the mechanisms used have not been disclosed and are the subject of a patent application.

ObjectStore provides an application program interface which programs are required to use in order to access and create persistent objects. Persistent objects must be created by a call to a procedure in this API. When using C, calls to the API must be made explicitly, in C++ the ability to overload operators and functions allows the operation *new* to correctly call the API as appropriate, presenting the programmer with the textual illusion of transparent object management.

2.5.2.1.2. Garbage Collection

C++ does not provide for automatic storage allocation or deallocation, and systems based on this language do not provide intrinsic garbage collection. ObjectStore's database mechanism provides persistence by reachability and as such some mechanism is needed to recover space within databases, however from available documentation the mechanisms used are unclear. It is apparent that in order to implement the translation of pointers described above that some mechanism for locating pointers within pages does exist within the system. It is possible that a mechanism similar to that used by Texas is employed.

2.5.2.2. Texas

Wilson and Singhal [Wilson 1991; Singhal, Sheetal et al. 1992] describes a system in which C++ programs execute in a persistent virtual address space. Of particular interest is the mechanism used to translate pointer representations between those used in the store and those used in addressable memory. This topic and the Texas system is covered in detail in the next chapter. Here we present a summary of the attributes of the Texas system.

2.5.2.2.1. Data Movement

Data movement in Texas is mediated by the virtual memory system, an attempt to access data which is not resident in addressable memory triggers a page fault and the store supplies the data. Texas translates the pointer representations between the store and addressable memory in such a way as to provide the illusion of a larger virtual address space than is actually supported by the machine hardware.

2.5.2.2.2. Pointer formats

The Texas system provides support for C++ programs. C++ does not provide a canonical object format and it is not possible to determine the location of pointers within an object by examination of the object alone. However C++ objects contain the identity of their class and from this it is possible to determine an object's format. Texas uses a post-processor which extracts the object format from the debugger support information created by the various compilers used. The debugging format descriptions are standard on most operating systems and thus Texas is able to host programs compiled with different compilers with no change.

2.5.2.2.3. Address Creation.

Texas programs only ever use the virtual addresses supported by the host architecture. The store system and swizling mechanism described in the next chapter ensure that store addresses are never visible to the running program.

2.6. Casper

The Casper system [Vaughan, Schunke et al. 1992] was implemented by a research group including the author to provide a distributed persistent system based upon a model of distributed shared memory mediated by page based virtual memory mechanisms. This chapter does not address the distribution of the system, this is covered in Chapter 6, but rather describes the mechanisms used to provide support for the Napier88 language.

2.6.1. Integrated Store Management

The page based systems described above are implemented above existing commercial operating systems. Systems which do not provide intrinsic support for orthogonal persistence and in which the abstractions provided are far from ideal. A major limitation with these systems is the inability to divorce the object movement from the management of the demand paged swap space. Considerable advantages accrue if the virtual memory manager is integrated with the persistent store.

2.6.2. Data Movement

The Mach [Acceta, Baron et al. 1986] operating system provides a mechanism by which user level programs can directly access the internals of the demand paging mechanism. In Mach it is possible to associate a region of a virtual address space with a service facility termed an *external pager* [Rashid, Tevanian et al. 1987]. An external pager is a user level

program which conforms with a system defined interface and provides the services normally provided by the operating systems paging mechanism. That is, it is responsible for the provision of data in response to page faults incurred within the region of virtual memory it services, is responsible for data removed by the operating system from that region (perhaps in response to pressure upon physical memory use), and is responsible for the servicing of access protection faults within the serviced region. This mechanism is close to ideal for the provision of shadow paging required to provide a resilient persistent store.

The distributed Casper system was constructed utilising the external pager mechanism. A significant advantage of this implementation is the integration of the persistent store with swap space. Utilising the external pager Casper completely integrates persistent storage with swap space.

Conventional systems retain data in swap space when available physical memory becomes depleted. Data may be required to migrate from swap space to addressable memory before moving to the stable store, adding considerable cost to the implementation.

2.6.3. Object Formats and Pointer Detection

In page based systems, sub-systems may be presented with an arbitrary page address rather than an object address . In general, in systems without hardware supported tags it is only possible to determine the location of pointers within an object if we are provided with the address of the beginning of the object. Therefore some further mechanism must be provided to aid in the location of pointers.

One mechanism is the crossing map. Crossing maps have their origin in garbage collectors [Appel 1989; Kolodner, Liskov et al. 1989]. To make use of crossing maps two requirements are made of object formats:

- A mechanism must exist so that the pointers within an object, and the size of the object can be found once the address of the head of the object is found.
- Objects must be allocated through memory, in such a manner that once one object is found, it is possible to chain forward to the next object in memory. Contiguous allocation is a simple mechanism to achieve this.

To find the pointers resident in a page it is only necessary to find some object that is located at an earlier address in memory and move forward from that object until the objects in the designated page are encountered. The simple crossing map uses this mechanism by

tagging each page with a single bit designating whether an object starts at the beginning of the page or not. Such a mechanism is represented in Figure 12 below.

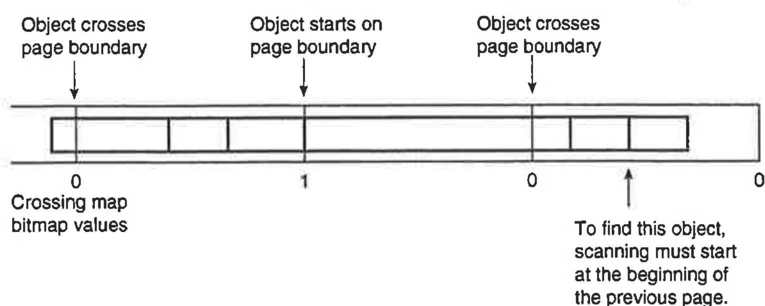


Figure 12. Using a single bit per page to represent object crossing.

A degenerate form of crossing map is to ensure that all page boundaries are coincident with object headers. This is possible if a special padding marker is used that signals that the reminder of a page contains no useful objects and that the next object can be found at the next page boundary. Objects larger than a single page still require special treatment, dramatically complicating such a scheme.

The use of a single bit to describe the crossing map has the disadvantage that the scan of objects may need to begin some pages before the page of interest. In systems for which the techniques were developed it is reasonable to expect that most of the pages are resident in addressable memory since they form part of the active working set of the program. In persistent systems, pages may require scanning for pointers when they are loaded from the store. There is no guarantee that the intervening pages will be resident in memory, however they must be brought from the store to allow the system to find the intervening objects and thus find the pointers on the needed page. Significant cost may be attached to fetching these extra pages from the store, pages which may otherwise be unnecessary for the running computation. A scheme where object allocation is padded to page boundaries helps considerably, but not for regions containing large objects.

A possible improved technique might be to maintain an array of pointers, containing one pointer for each page in the system. Each pointer points to the first object header before or aligned with the start of the page. In this way, at most two pages (the required page and the page upon which the start of the first object resides) need to be examined when a page is faulted.

If pointers are stored contiguously in objects, an improved scheme is possible. This is the scheme introduced by the Casper system. Rather than an array of pointers, an array of tags is maintained, with each tag corresponding to a page in the store. Each tag describes any partial object which may overlap the start of the page. The tag consists of:

- the offset of the first pointer in the partial object
- the number of pointers in the partial object on that page
- the number of scalar values in the partial object on that page

Objects constructed using the format used by the PAM may be easily encoded into a 32 bit word. Since a PAM object header is two words in length two bit suffice to encode all possible offsets to the pointer fields. The remaining 30 bits allow 15 bits each for encoding the number of pointers and number of scalars in the first object, this allows for systems with pages of up to 64 kilobytes in length to use this format. The format and some possible page boundary crossings and their encoding are illustrated in Figure 13 below.

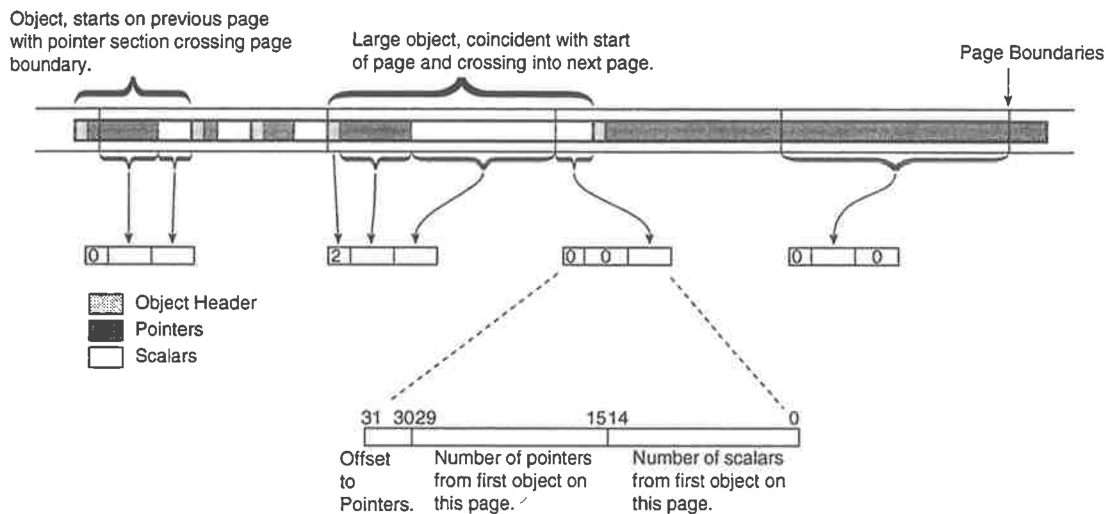


Figure 13. Representations of different object crossings.

This mechanism requires that only the faulted page and its tag need be examined when a page fault occurs. As desribed in Chapter 4, tags may be kept with the page translation entries used to describe the page's location within the store, thus the tag will be available when the page is read from the store.

When new objects are created or when objects are moved (such as when garbage collection is performed) the system must check whether the object crosses a page boundary and update the crossing map. As is described later, the Casper system maintains a distinct address range (the local heap) in which objects are allocated and the majority of execution occurs. To improve execution speed the system does not eagerly update crossing map entries for the local heap, rather these crossing maps are only updated when the heap is copied to the persistent store.

An important ramification of this optimisation is that native code generation is possible within this system without the need to perform any crossing map operations. In the Napier88 code generation system [Bushell, Dearle et al. 1994] generated code has no

knowledge of the crossing map mechanism, and is not required to perform any action in its support. This simplifies the generated code reducing both its size and execution time.

2.6.4. Address Generation

In the Casper system, the address of an object is the virtual address at which the object resides. These addresses remain the same in the store. Objects become resident through the movement of, and are accessed upon, the virtual memory page that they are placed.

This limits the size of the system to that of the underlying machine architectures virtual address space. Chapter 3 discusses how to extend this limitation through the use of swizzling techniques. Chapter 5 discusses the internal operation of the Casper store.

2.6.5. Local Heaps and Pointer Quarantine

The Casper system provides each process in the system with a region of contiguous memory in which to allocate objects, this serves to restrict computation to within this region as much as possible, and also acts as the most volatile generation of a generational garbage collection scheme. In the distributed version of the Casper system local heaps provide segregation of pointers between processes, considerably simplifying garbage collection. We term the mechanisms that segregate pointers *pointer quarantine*.

2.6.5.1. Distributed Casper

The distributed Casper system is described in detail in Chapter 6, however for the purposes of describing the copy-out system the following will suffice.

The Casper system implements a distributed shared memory (DSM) model in which individual clients share a single persistent virtual address space. A central server is responsible for stable storage. Clients gain access to the data in the store by making requests of ther server. Clients and the central server co-operate to maintain coherency of the separate instances of data each client.

The distributed Casper system provides each separate process with a local heap. Each local heap is a separate contiguous range of addresses taken from the shared persistent virtual address space. When executing, each process resides within a different client. To avoid greatly complicating garbage collection of local heaps, the system enforces a regime in which only pointers within objects resident in a local heap may refer to other objects in that local heap. The maintenance of this invariant is clearly deeply tied into the maintenance of a generation garbage collection mechanism. This management system is

known as the copy-out system. Legal and illegal pointers in the Casper system are depicted in Figure 14 below.

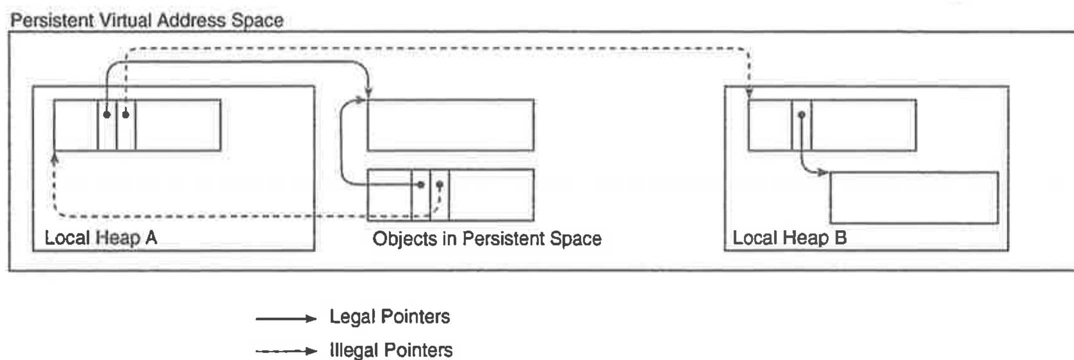


Figure 14. Legal and illegal pointers in Casper.

2.6.5.2. Copy Out Implementation

If a client attempts to access a page which it does not currently hold a copy of, it will request a copy from the central server. If the server determines that the most up to data copy is held by another client it will forward the request to that client. This second client is required to make available a copy (for read access) or yield the page (for write access) to the client originating the request. Before supplying this page the yeilding client must ensure that the page does not contain any pointers that refer to objects within its own local heap. To achieve this, at least the immediate closure of such pointers must be copied from within the local heap to some area of virtual memory outside any processes local heap.

Operation of the copy-out system has close links with the operation of generational garbage collection systems, although the triggering mechanisms are different.

2.6.5.3. Eager Copy-Out

Initial implementations of the Casper system instituted an eager copy-out regime. As soon as a pointer was created in an object outwith a client's local heap referring to an object within the local heap, the support copy-out system would copy the entire transitive closure of the pointer to a region outside of the local heap. This implementation was written partly as a matter of expediency, however the ramifications of the policy are deserving of study.

Programming in the Napier88 system has evolved a style in which complex data structures are assembled and then as a final act, linked into the persistent environment. Such a style fits well with an eager copy-out system. Programs written to load the persistent stores with the initial environment run with little performance penalty from an eager policy.

When running in an interactive system an eager policy fails spectacularly. The interactive system and window manager links a large number of data structures for the representation of a session into the persistent environment. These structures are instantly copied out of the local heap, the program essentially paying the cost of creating each object twice. These objects include large data structures such as the persistent representations of the user's frame buffer and each individual window. All these structures will become unreachable when the interactive session is terminated and thus clutter the store, only being removed by a store level garbage collection. Ameliorating this somewhat, interactive sessions are persistent and may have a considerably longer lifetime than conventional interactive sessions. Indeed a user could potentially keep the same session for their entire lifetime, reconnecting to it whenever they need.

Another unfortunate habit of the eager system is to occasionally copy the entire closure of the running process out of its local heap. This happens because procedure closures are first class objects and it is quite reasonable to link one into the persistent environment. Such an occurrence destroys all the advantages of the local heap.

2.6.5.4. Lazy Copy-Out

Clearly an eager policy is of only passing interest. Generation garbage collection systems usually implement some form of tenuring policy in which an object must survive a number of garbage collection passes before being promoted in to the next generation. This is not possible in the Casper system since the need to move an object from within the local heap is the result of the action of other clients in the system. However there is no need to perform the movement of objects within the local heap until a request for a page containing references to these objects is received.

Furthermore the copy-out system need only copy those objects directly referenced to outside the local heap. These objects may themselves later cause further movement of objects should the page onto which they are copied become shared.

When a page that has been used to receive copied-out objects itself becomes shared the system attempts to fill any remaining space within the page with local objects reachable from objects already resident in the page. This has the effect of naturally clustering objects onto pages; locality of placement in memory follows reachability between objects. The movement of objects from the local heap into shared pages provides the natural place to effect clustering mechanisms, clearly more sophisticated tactics could be used than the one described. Eager filling of shared pages also has the effect of making available some of the

closure of the requested objects, so that further requests for pages containing these referenced objects can be circumvented if they occur. Obviously the value of this tactic depends upon the access patterns of the requesting client.

Upon the death of a program the system copies out those objects within the local heap that are still reachable from outside the local heap. These also become clustered together.

2.6.5.5. Page/Card Based

Rather than maintain a remembered set through explicit test on pointer assignment it is possible to utilise the page protection mechanisms to provide tracking of pointer assignments [Appel and Li 1991; Hosking and Hudson 1993]. This mechanism is akin to the hardware maintained page tags provided by the Symbolics 3600. It operates as follows.

Pages within the local heap are freely modified, those pages outside the local heap are protected against write access. Whenever an attempt is made to write to a protected page a user level exception handler is invoked. This handler records the page's address and changes the protection on the page to allow write access, then allows the excepting code to continue. The list of modified pages forms a surrogate for the remembered set. Unlike the Symbolics 3600 the system has no way of determining whether a write operation to a page actually involved a pointer assignment or indeed whether the pointer referenced a younger generation, the system pessimistically assumes that any write operation creates such a pointer. When the local heap is garbage collected, all the recorded pages are scanned to find any pointers that refer to objects within the local heap, thus building a remembered set. A mechanism is needed to find pointers within the listed pages from only the address of the page, the crossing map systems described earlier provide this ability.

A valuable result of using this mechanism is that the mutator code need have no knowledge of the operation of the copy-out mechanism. There is no need for an explicit test upon pointer assignment. This is especially valuable when generating native code as it considerably simplifies the code generated, improving execution speed and reducing code size. The native code generation for Napier-88 [Bushell, Dearle et al. 1994] is implemented above this system and is able to execute oblivious to the operation of the copy-out system.

2.6.6. Store level garbage collection.

Store level garbage collection remains a taxing subject no matter what the implementation chosen for data management. Garbage collection of persistent stores suffers from two problems.

- The sheer size of the store and data to be traversed.
- The overhead of extra storage needed to reflect the massive changes that occur in garbage collection.

Schemes to garbage collect a stable store will often need to traverse a data set many times the size of their virtual memory working set and need to use large quantities of stable store. Algorithms used need to take into account these extra constraints. The Casper system provides two mechanisms for reclaiming space in the store.

2.6.6.1. Opportunistic memory recovery

A system which implements a page based persistent virtual address space can implement a crude but useful mechanism to reclaim some space without requiring much effort. By traversing the closure of the store it is possible to build up a list of all pages that contain reachable objects. Those pages that contain no reachable objects are simply deallocated.

The opportunistic system traverses the transitive closure of the store marking objects. However marking the objects themselves would require the creation of shadow copies in the store and the allocation of extra storage, one of the costs we wish to avoid. Instead a local mark table is maintained in volatile memory, implemented as a hash table indexed by address of object. An in-memory copy of the virtual memory allocation map is updated by removing each page from the table when any part of a reachable object is found upon it. The remaining pages are candidates for deallocation.

At first sight this system would appear to offer only limited opportunities for the reclamation of space. However the nature of the copy-out system maintains locality of connected data and tends to cluster unreachable sub-trees within contiguous memory. Furthermore the mechanism is able to free the space occupied by very large garbage objects such as image data, persistent representations of windows, and frame buffers. Freeing a single colour frame buffer alone reclaims one megabyte of store.

2.6.6.2. Full Garbage Collection

The second mechanism performs a complete compacting garbage collection of a Casper store and will free all unreachable objects. The principle design objective of the system is to reduce the number of times that random access to the store is needed, and wherever possible to access stores in address order, or better, in file block order. It operates as follows.

2.6.6.2.1. Mark Phase

The mark phase traverses the closure of the store, similarly to the opportunistic collector, it does not mark objects themselves but rather maintains an internal, volatile data structure. This structure lists the address of each object found. This phase unavoidably results in random access to the store as it traverses the closure. However as we shall see this is the only time random access is required.

In the current version of the collector the list is maintained as a bit-map where each bit in the map corresponds to a four byte word in the store. Since this bitmap is logically quite large (one 32nd of the size of the memory space represented by the store) it is structured heirarchically, using a two level description, and only allocates memory for those areas of the bitmap that are actually used. A bitmap is particually advantageous when the store contains many small objects. If the average object size is 128 bytes then the bitmap will occupy a similar amount of space as a hash table, once the average object size falls below this the bitmap performs better. More importantly, the bitmap provides an iterator which provides the reachable objects in address order. This feature is used in the next phase.

2.6.6.2.2. Translation Phase

Once the mark bitmap is complete it is travered in address order. As each reachable object is found an allocation is made in the new store for the object. This is acheived by reading the object size from the store and incrementing an allocation pointer; effectively this is a simulation of the compaction phase. Although the store is traversed a second time, since the traversal is performed in address order the virtual memory and file sub-systems can make use of the access order to improve performance. Rather than copy the object immediately, a translation table is constructed. This is an array of entries each containing: the original address and, new address of each reachable object. It is constructed as the translation phase progresses and thus entries are inserted into the array in order of original object address. Once the translation table is complete the compaction phase can proceed.

2.6.6.2.3. Compaction

The garbage collector does not compact into the existing store, but rather a new store is created to receive the data. This has the advantage that the new store can be accessed in linear order of physical pages (being empty, it is free to use any order of physical pages), thus improving access speed. A new store stub is created utilising the store description data in the header of the current store. As the compaction phase progresses the data storage for

the store is allocated. Store structure and the mechanics of store creation are covered in Chapter 5.

Once a new store is ready, the compaction phase can begin. The original store is traversed in address order. Each object encountered that is present in the mark bitmap is copied into the new store, allocating store space as it proceeds. The copy process translates the pointers in the copied objects as it proceeds, utilising the translation table. Since the translation table is ordered by original address, entries can be found using a binary search. Thus the compaction phase proceeds with a linear address traversal of the original store and writing of data to the new store in linear physical block order.

Once complete the original store can be discarded or cleared for reuse.

2.6.6.3. Summary of Garbage Collection

The garbage collection systems described are designed to avoid random access to the underlying store where possible and to perform with a small working set. However, the volatile data structures may require a significant allocation when stores become very large. This is an ultimate limitation to the scaleability of the mechanisms.

The opportunistic collector is intended to free clumps of the persistent virtual address space quickly. Since it neither compacts objects in the address space, nor is able to free all unreachable object allocations its utility is limited. However it runs quickly relative to the full collection mechanism, and does not require the allocation of extra stable storage.

The full compacting collector frees all unreachable objects and compacts the address space. Since it creates a new store it is able to place data onto file blocks in block order. This both improves the speed of writing whilst the mechanisms executes, but also acts to re-cluster contiguous addresses in the persistent space onto contiguous file blocks, which in general will improve the performance of subsequent uses of the store.

2.7. Performance

Evaluating the relative merits and deriving metrics of performance of the systems described above is a difficult problem. In almost all cases the systems are research vehicles, often aspects of performance are considered peripheral, or the resources to create optimised implementations is lacking. When efforts are expended in improving performance it is hard to judge how much of a systems performance is due to the inherent merits of the overall design versus the amount due to the specific performance enhancements made.

Whilst a depressing scenario, considerable work has been performed in attempting to characterise the relative merits of some implementation strategies.

When attempting to evaluate the relative merits of pure software implementations, as represented by Mneme and the PAM versus implementations that seek to take advantage of the features of the underlying hardware no simple relationships exist. In essence hardware systems attempt to amortise the higher cost of handling page faults and data access exceptions through the elimination of many smaller and lower cost run-time checks. Work carried out by Hoskings and Moss has begun to characterise these trade-offs. Their work [Hosking and Moss 1993] is based upon a DEC workstation utilising a MIPS R3000 processor and instrumented by a high resolution timer supplied by DEC. This has enabled them to make very fine and accurate timing of the execution of the Mneme system, and allowed them to compare the standard Mneme system with version that make use of some aspects of page based exception handling instead of runtime tests.

2.7.1. Cost of exceptions

Both Appel and Li, and Hoskings and Moss identified the intrinsic cost of exception delivery as a limiting factor in the utilisation of page based virtual memory architectures for the roles described above.

The cost of exception delivery is, in general, very high in current systems. This expense is not an intrinsic part of the processor architecture (although some are worse than others) but rather a product of an operating system design wherein exceptions are considered to be just that, exceptional; of sufficient rarity that highly general but expensive mechanisms are appropriate.

2.7.1.1. Exceptions with CISC architectures.

Complex instruction set architecture implementations are complicated by the need to handle exceptions. If an exception occurs in mid-instruction the instruction must be restartable. This requires care in the design of the micro-code and the availability of enough state information to perform the restart. The 68020 for instance, places a 60 word descriptor atop the processes kernel stack in response to an exception. This descriptor contains the register state at the time of the exception, information relating to the nature of the exception and the contents of internal registers. This descriptor block is reloaded when the excepting process is resumed, allowing the process to resume correctly. Managing this

state information is clearly a performance impediment. The cost of writing and reloading can amount to some hundreds of machine cycles.

2.7.1.2. Exceptions in Berkeley RISC.

In a Berkeley RISC architecture (such as the SPARC) a stack of registers is maintained, each function instance is allowed the use of a new set of registers (a *window*) allocated from this stack. Parameter passing is achieved by overlapping part of the windows. A function call is saved the cost of memory store and load operations to the in memory stack and can reap significant performance benefits. However if the space of register windows is exceeded (an overflow) a previously used window must be flushed to memory, freeing space for a new allocation.

When an exception occurs on such an architecture the entire set of windows must be saved to memory and reloaded after the exception is handled. When executing in user mode flushing a single register set to memory takes about 60 processor cycles [Henessy and Patterson 1990], however when handling an exception the task is significantly more complex. Whilst flushing the register set to memory, the system must ensure that another exception cannot occur. In particular this precludes flushing to virtual memory, since a page fault could occur. Instead, the operating system emulates virtual memory explicitly, adding greatly to the cost of handling an exception. Handling exceptions with the SPARC architecture requires upwards of 1,000 machine cycles [Irlam 1992] before any operating system functionality is added.

This added cost clearly limits the use of an exception model on such architectures.

2.7.1.3. Exceptions on conventional RISC architectures

We term a RISC architecture without register windows as conventional. Exception handling on such architectures is in general fairly straight forward. The kernel only need save a small subset of the registers and then invoke the users exception handler.

These architectures may suffer a general performance penalty in comparison with those endowed with register windows, Henessy and Patterson [Henessy and Patterson 1990] noted that their DLX architecture required 1.6 times as many store operations as the SPARC on simple C benchmarks. Compiler assistance in the form of inter-procedural register allocation (whereby parameters are passed in registers between calls) helps to close this gap. However because of the nature of the late binding and incremental construction used in persistent languages inter-procedural register allocation is not always possible. It is

not clear that register windows are of the same value in such languages, ameliorating the deficit. Further, current realisations of Berkeley architectures appear to be unable to reach the processing speeds provided by well designed conventional RISC architectures. This is a highly volatile area and it is difficult to draw useful conclusions.

2.7.1.4. Exceptions in Unix implementations.

Unix presents an exception handling model which combines true exceptions (those that occur synchronously with program execution) with an out of band signalling mechanism and process control. A process receives an exception as a *signal* which is delivered by the automatic instantiation of a function atop its current execution stack. The system is capable of receiving recursive signals (that is a signal handler may itself incur exceptions) although by default reception of the same kind of signal is blocked whilst handling the original signal.

2.7.1.5. Exceptions in OSF and Mach.

The Mach operating system has proven to be an important vehicle in the implementation of page based persistent systems. It also forms the basis for the implementation of OSF/1, a commercially important Unix implementation. An important attribute is a very flexible exception delivery mechanism. Exceptions may be delivered as a message to a separate service program which can handle the exception whilst the original thread of execution remains blocked. The virtual memory system is similarly handled in a flexible manner; page faults and access exception faults may be handled by a separate handler at user level. This flexibility comes at a considerable performance cost. When an exception is generated the kernel manufactures a message containing a description of the exception. This message is enqueued onto a message port for reception by the Unix emulation layer. This layer is later activated and receives the message, synthesises the Unix signal delivery and when the signal has been handled the process is reversed. Thus exception delivery requires the delivery of messages, and a number of context switches to effect.

2.7.1.6. Measuring Exception Cost

The work by Appel and Li, and Hoskings and Moss has been directed at measuring the cost of exception delivery and virtual memory sub-system under Unix. This is important as Unix provides the operating environment for the majority of implemented persistent systems. However it is equally important to notice that the cost of exception delivery under these Unix implementations is considerably poorer than it need be. Hoskings and Moss

noted that the effects of processor caches can have a distinct effect upon exception speed, exceptions occur very frequently having a significant advantage.

When considering the utilisation of exception for the support of the mechanisms described earlier four costs must be considered

- the intrinsic cost of an exception,
- the cost of delivery of the exception to the user,
- the cost of the kernel internals in interpreting the exception, and
- the cost of systems calls to set up appropriate page protection.

The first of these is limited by the machine architecture, the second by the operating system design. The third item is for the purposes discussed, a measure of the kernel's speed in interpreting the virtual memory description structures. The last item is partly a combination of the first and third items.

Appel and Li measured the cost of protecting a page of memory, taking an access exception upon that page and then reducing the protection upon the page. Their results showed a variation of up to six times the number of machine instructions needed to perform this task across different Unix implementations.

2.7.1.7. Building Exceptions for Speed

The MIPS architecture is able to provide for the delivery of exceptions directly to user code, bypassing entry into the kernel. (This is not strictly accurate for virtual memory faults as will be noted below, however for other forms of exceptions this is true.) Unfortunately this feature is not used in Unix implementations as it does not conform to the Unix model. Recently Chandramohan and Levy [Chandramohan and Levy 1994] have proposed fast exception handling mechanisms that do the majority of processing of the exception delivery in user space and avoid one of the two system traps inherent in conventional designs.

In Chapter 7 the architecture of a new operating system (the Grasshopper operating system) will be described. One feature of the design of this operating system is the provision of fast handling of exceptions, the design being tailored to make the implementation of the features described earlier in this chapter efficient. Early figures from implementation suggest that exception delivery to the user level is some 20 times faster than that of OSF/1 on the same hardware and remains considerably faster than other Unix implementations on all platforms.

2.7.2. Page Granularity

One of the most obvious differences between page based systems and those based upon object movement is the granularity of data. This becomes important when the costs of moving data to and from stable storage is considered. In principle object systems only need move the precise amount of data needed to represent an object whilst a page based system is constrained to move at least one entire page at a time.

In principle all stable stores are built using a block oriented device, all disk drives store data in fixed length sectors. The majority of disks use sectors of 512 bytes, however the use of intelligent controllers with local caches and track buffers makes it more efficient to transfer larger amounts of contiguous data at a time. Indeed the current page sizes used in virtual memory are if anything small compared to the most efficient data transfer sizes. Those store architectures that do support individual object access put considerable effort into clustering of objects on disk blocks. Systems that present a persistent virtual address space achieve essentially the same gains by clustering objects in virtual memory appropriately. However there remain significant differences in performance between systems that provide for individual object movement and page oriented systems. A particular area is in the nature of efficiency of transactions.

Hoskings and Moss measured variants of the Mneme systems that provided for either individual object movement or page based movement. If transactions are small and frequent only a small number of objects will require copying to stable store. If the store supports logging of changes only a small amount of data need be transferred to disk. If transactions are long and involve considerable modification of resident data the costs of determining the modified data and preparing the log begin to dominate and page based stores are superior. In the Napier88 systems described transactions are not supported (although recently Dave Munro [Munro 1993] has described a page based store in which transactions are supported for Napier88.) In general, in Napier88 programming the nature of computation is more oriented toward long periods between the generation of stable states in the store.

2.7.2.1. Page cards vs. remembered sets:

Hoskings and Moss [Hosking and Moss 1993] produce figures which compare the costs of managing pointer quarantine for generational garbage collection. They compare page protection with software cards and a remembered set system. They also provide test results

for a page based system in which the identity of the modified pages are known in advance and thus can simulate a system in which the cost of exceptions is zero.

They presented two benchmarks, a destructive tree modification which generates large amounts of garbage, and an interactive benchmark which simulates interactive activity with a Smalltalk system.

In a system implementing a remembered set system the destructive benchmark spent about 7% of the time checking and entering entries into the remembered set. A page based card system used about 8% in servicing exceptions and marking pages. However the extra work involved in traversing the pages to find roots of reachability (some 5% extra) reduced the overall performance of the page card system. On the interactive benchmarks the overall performance was very similar. It seems difficult to make any firm conclusions about the relative merits of either scheme. Improvements in the speed of exception delivery can only reduce the impact of an activity that currently only accounts for 5% of the running cost.

The above work was done on an interpreted Smalltalk system. Hoskings and Moss suggest that when native code is used the costs of garbage collection will begin to increase in importance as garbage collection will not increase in speed. Thus the costs of page traversal will grow in importance, swinging the balance in favour of remembered sets and run-time checks. In support of a page cards system we should note that when native code is used the overhead of the interpreter is removed and the costs of the runtime checks will also become more important. Native code generation is also considerably simplified when it does not need to incorporate such tests. The current native code generation for Napier-88 [Bushell, Dearle et al. 1994] achieved significant speed improvements simply through the removal of the interpretation control loop and overhead of parameter marshalling. A doubling of performance through such improvements will double the impact of run-time tests for pointer assignment on system performance.

2.7.2.2. Object Residency Test

Similar arguments can be made for the costs of object residency tests. However there is a considerable amount of flexibility available to the designers of systems when object movement is implemented. Earlier we have reviewed systems which use individual run-time checks, utilise exceptions to trigger object fetching, use the method dispatch mechanisms to short circuit the run-time test, use run-time checks but read from a memory mapped store through to pure page based systems.

Hoskings and Moss also compared the relative costs of using exceptions to trigger fetching with method dispatch. Their results confirm method dispatch as a highly efficient mechanisms for those systems which can exploit it. They found that exception based triggering performed significantly poorer. However this is in a system in which the store system is distinct from the object residency management.

When a system utilising either explicit tests or method dispatch, fetches objects into addressable memory it will often incur a page fault because the area of memory into which the data is to be placed is, until then, unused and no physical memory is mapped. Fully integrated page based systems, those in which the provision of swap space is integrated with the stable store do not suffer from this since the provision of mapped physical memory occurs as part of the faulting mechanism. Such integration is possible using the Mach External Pager mechanism described in chapter 4 and is also provided in the Grasshopper operating system described in Chapter 7.

Similar arguments about the relative impact upon native code upon comparisons between exception based systems and those employing run-time residency checks can be made. As the code becomes more efficient the costs of run-time checks will become more important.

2.8. Comparisons.

Table 1 below presents a comparison of the major issues described for the discussed systems. The table compares the following design features:

- Orthogonal PS: whether the system presented can be considered to provide persistence as an orthogonal attribute.
- PS Model: the model of persistence provided by the system.

  R = Persistence by reachability.

  VM = Persistent virtual address space.

  File = Persistence through the file system.
- Separate Addrs: whether the system maintains separate addresses in addressable memory and the persistent store.
- Object Format: The manner in which objects are described an recognised by the various sub-systems.

  M = Multiple, separately coded object formats.

M/P = Multiple separate formats plus pointer CDR coding.

C = Single canonical object format.

Sym = Object formats determined from complier symbol tables.

- Crossing Map: whether the system uses a crossing map to locate the beginning of objects.
- Pointer Location: the manner in which the system identifies pointers.

  Tag = The hardware provides segregated tag bits.

  ST = The system implements software tagging of pointers.

  Obj = The system identifies pointers from the object format.

  Sym = The system uses compiler generated format information.

- Access Protection: the manner in which the system is able to protect data from illegal or unsafe access, it the system is able to.

  HW = The system provide hardware enforced access control.

  SW = The software enforces access control.

- Object Table: the system utilises an object table to describe the contents of addressable memory.
- Auto Data Alloc: the system supports automatic allocation of data.
- Data Moved By: the mechanism by which data is moved to and from the persistent store.

  M = Microcoded special instruction.

  OS = Data is moved by direct action of the operating system.

  U = Data is moved by code executing in the user space.

- Move Trigger: the mechanism which triggers the data movement system.

  H = The system hardware directly causes the code to run.

  E = A system exception mechanism triggers data movement.

  UT = A user level test is required to begin data movement.

- PID Bits: the number of bits available in a persistent object identifier.
- HW Addr Bits: the number of bits notionally available on the host hardware to address data.

- Integrated Swap: whether the system is able to directly evacuate objects from addressable memory to the persistent store to avoid pressure on *physical* memory.
- Garb Coll: whether the system supports garbage collection to free either addressable or persistent memory.
- Page Cards: whether the system utilises a page protection mechanism to implement generational garbage collection. (Opt = optional.)
- Rememb Set: whether the system implements a remembered set to support generational garbage collection.

  ML = That the system effectively provides a remembered set through the maintenance of a list of modified objects that must be retained.
- Store GC: whether the system provides a mechanism to separately garbage collect the persistent store.
- Explicit Commit.: whether the system both provides and requires a special function that copies the system state into the persistent store.
- Language Spec: whether the system is specifically designed for the use of a particular language.
- Native Code: whether the system is currently able to support the use of code generated native to the underlying hardware.
- No Checks: whether, if native code were generated, the code could execute without the need to check for constraints imposed by the support system.

	Hardware				Software					Page Based		
	Rekursive	Symbolics 36000	Monads	ORSLA	Smalltalk OOZE	Smalltalk LOOM	Smalltalk Mneme	PS-algol CPOMS	Napier-88 POMS	Texas	ObjectStore	Napier-88 Casper
Orthogonal PS	Y	N	Y	Y	N	N	N	Y	Y	N	N	Y
PS Model	R	File	VM	VM	R	R	R	R	R	VM	VM	VM
Separate Addrs	N	N	N	N	N	Y	Y	Y	Y	N	N	N
Object Format	M	M/P	-	M	C	C	C	M	C	Sym	Sym	C
Crossing Map	N	Y	-	N	N	N	N	N	N	Y	N	Y
Pointer Location	Tag	Tag	-	Tag	ST	ST	ST	Obj	Obj	Sym	Sym	Obj
Access Protect	HW	HW	-	HW	SW	SW	SW	SW	SW	-	-	SW
Object Table	Y	N	-	N	Y	Y	Y	Y	Y	N	N	N
Auto Data Alloc	Y	Y	-	Y	Y	Y	Y	Y	Y	N	N	Y
Data Moved by	M	OS	OS	OS	U	U	U	U	U	OS	OS	U
Move trigger	H	E	E	E	UT	UT	M	UT	UT	E	E	E
PID Bits	38	32	128	50	15	31	31	31	31	64	32	32
HW Addr bits	38	32	128	50	16	16	32	32	32	32	32	32
Integrated Swap	Y	N	Y	Y	Y	Y	N	N	N	N	N	Y
Garb Coll	Y	Y	-	Y	Y	Y	Y	Y	Y	N	N	Y
Page Cards	N	Y	-	N	N	N	Opt	N	N	-	-	Y
Rememb Set	N	N	-	Y	N	N	Y	ML	ML	N	N	N
Store GC	Y	N	-	N	N	N	N	Y	Y	N	N	Y
Explicit Commit	N	N	N	N	N	N	Y	Y	Y	Y	Y	N
Langage Spec	Y*	Y	N	N	Y	Y	Y	Y	Y	N	N	Y
Native Code	Y*	Y	Y	Y	N	N	N	N	Y*	Y	Y	Y
No Checks	Y	Y	Y	Y	N	N	N	N	N	Y	N	Y

Table 1. Comaprison of salient features.

2.8.1. Notes.

- The Rekursiv is considered to be language specific because it is necessary to provide a completely separate microcoded instruction set to support new languages.
- The Rekursiv is considered to support native code generation by virtue of its microcoded instruction set, although the complexity of the environment provided and the restrictions to a single language per instruction set make this classification somewhat equivocal.
- The Napier88 POMS implementation is listed as supporting native code generation since a native code generator is currently under construction.

Chapter 3. Pointer Swizzling

3.1. Introduction

Most persistent and database programming languages are supported by an object store, a conceptually infinite repository in which objects reside. In order to manipulate these objects, they must be fetched from the object store into directly addressable memory, usually virtual memory. In systems which support orthogonal persistence [Atkinson, Bailey et al. 1983] this is performed transparently. Thus in these systems, two different kinds of object addresses may exist: those in the backing store (persistent identifiers or PIDs) and those in directly addressable memory (virtual addresses).

Many researchers have argued that large pointers (anywhere up to 128 bits) are required to support persistent systems [Cockshott and Foulk 1990; Rosenberg 1991]. Persistent pointers need not be the same size as those supported by virtual memory (usually 32 bits); indeed persistent identifiers may be arbitrarily long. The persistent address of an object may be mapped onto a virtual address in a number of ways:

- Dynamically translate from a PID to a virtual address on each dereference.
- Make an object's virtual address coincident with its persistent identifier.
- Perform a once only translation from a persistent identifier to virtual address, overwriting the copy of the persistent identifier in the virtual address space with a virtual address so that all subsequent dereferences incur no translation penalty.

This last option has become known as *pointer swizzling* and is the subject of this chapter. The first option, dynamic translation, is seldom more efficient than swizzling [Moss 1991]. The second option is only possible if persistent stores are small enough to be contained within the virtual memory. All these techniques have been used to implement persistent object stores [Kaehler and Krasner 1983; Cockshott, Atkinson et al. 1984; Vaughan, Schunke et al. 1992].

Pointer swizzling may be performed at a variety of times, the earliest being when objects are loaded or faulted into memory; this is termed *eager pointer swizzling*. The latest time swizzling may be performed is when a pointer is dereferenced, and is termed *lazy pointer swizzling*. When swizzled objects are removed from virtual memory, virtual memory pointers must be replaced by PIDs; this is often referred to as *unswizzling* or *deswizzling*.

Eager pointer swizzling has some advantages; in particular, if a data set may be identified in its entirety, all the pointers may be swizzled at once, avoiding the necessity to test whether

a reference is a PID or a virtual address prior to every dereference. However, this approach has the disadvantage that pointers may be swizzled, involving some computational expense, and never used.

Some systems use an *ad hoc* swizzling scheme; in these systems persistent pointers are the same size as VM addresses and may be coincident with the virtual address space. Whenever possible data is simply copied at the appropriate position into the virtual address space from the store. However if the appropriate region has already been allocated, swizzling is employed. It is believed that a variation of this scheme is also used by Object Design [Lamb, Landis et al. 1991].

In persistent systems it is unusual to be able to identify a self contained data set and some lazy swizzling is unavoidable. As described in the previous chapter many persistent systems that employ swizzling rely upon a software test to distinguish between PIDs and local addresses. Recently, schemes have been described which avoid performing these tests by performing pointer swizzling at page fault time [Wilson 1991]. This chapter describes a hybrid technique which offers many of the advantages of both these approaches.

The remainder of the chapter is structured as follows: firstly we will describe a typical software address translation scheme. This is followed by a discussion of Wilson's scheme: a technique for performing pointer swizzling at page fault time. Next we introduce a new scheme which is a hybrid and performs swizzling in two phases and an analysis of this scheme is made. We also suggest some implementation techniques that may be utilised in conjunction with such a scheme. The chapter concludes with a comparison of the three architectures.

3.2. Software address translation

The first object systems to be called persistent [Atkinson, Chisholm et al. 1981; Atkinson, Bailey et al. 1984] performed lazy pointer swizzling implemented entirely in software. In this section, for illustration purposes, we will concentrate on one of these, the Persistent Object Management System written in C, the CPOMS [Brown and Cockshott 1985]. The CPOMS is the underlying system used to support implementations of PS-algol [PS-algol 1988] under Unix.

The persistent store implemented by the CPOMS is a large heap with objects being addressed using persistent identifiers (PIDs). PIDs may be arbitrarily large but in current implementations PIDs are identical in size to the normal pointers used by the PS-algol run time system [PS-algol 1985]. PIDs are distinguished by having their most significant bit set.

Hence it is possible for the PS-algol run time system to distinguish between a valid address in virtual memory and a PID.

PIDs are pointers to objects outside of the program's virtual address space, therefore the objects to which they refer cannot be directly addressed by a PS-algol program. To ensure that PIDs are not dereferenced, a test is made prior to the use of any object address; in the PS-algol system this test is made using in line code. When an attempt to dereference a PID is detected, the referenced object is fetched into memory and the PID is swizzled and replaced with the appropriate virtual address. This process is shown in Figure 15 below in which objects B, C and E have been fetched into directly addressable memory where they are represented by objects B', C' and E'. Note that some references within virtual memory are virtual memory addresses whereas other are PIDs.

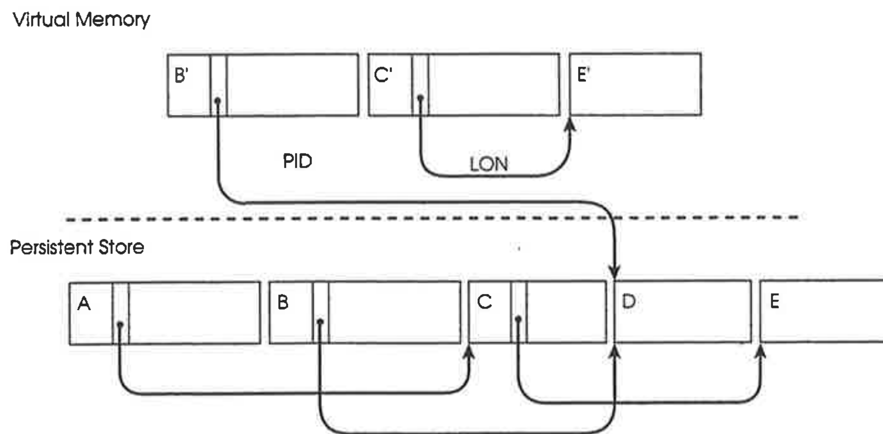


Figure 15. Swizzling in PS-algol

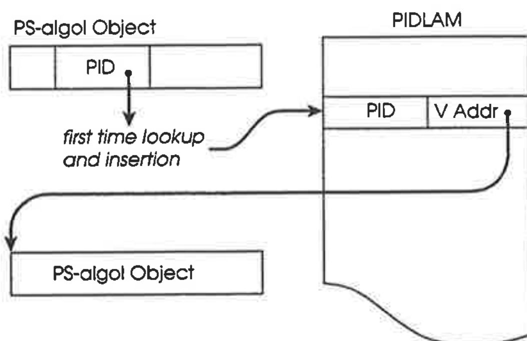


Figure 16. Looking up a PID in the PIDLAM

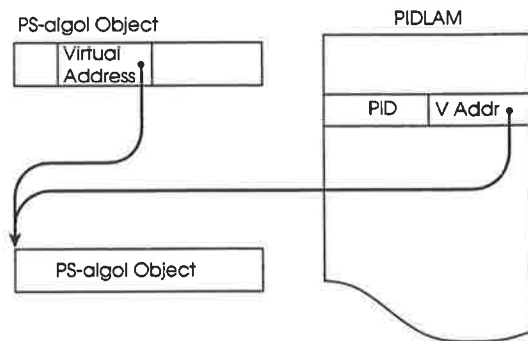


Figure 17. Overwriting a PID by a virtual address

The CPOMS maintains an Object Table (termed the PIDLAM (the PID to Local Address Map). When a PID is first used and the object to which it refers is copied into local memory, the PID is entered into the PIDLAM along with the virtual address of the copy as shown in Figures 16 and 17. Therefore, if another instance of the same PID is encountered, the

address of the copy can be found from the PIDLAM. This is necessary to preserve referential integrity in the running system.

Although relatively simple, this mechanism compromises performance in five areas:

1. all the address translation is performed in software,
2. all pointer dereferences must be checked using software to ensure that the pointer is not a PID,
3. disk fetches occur on a per object basis,
4. large objects must be copied into virtual memory in their entirety, and
5. every unswizzled pointer to an object must be swizzled at the time of dereference, even if the referend is resident in local memory.

The first, fourth and last of these problems may be eliminated if the hardware address translation mechanisms can be exploited. As stated earlier, this is only possible if the persistent identifier of an object is made coincident with its virtual address; clearly this approach results in relatively small stores on 32 bit architectures. The second problem may be eliminated if persistent addresses are illegal virtual memory addresses since an access will cause the hardware to raise an exception. This is only more efficient if the operating system provides a light weight exception mechanism. The CPOMS partially addresses this problem by eagerly swizzling certain pointers and in so doing avoids some checks. For example, pointers loaded onto a stack in the dynamic call chain are eagerly swizzled. The third problem may be overcome by amortising the cost of disk access across many object fetches.

3.3. Address translation at page fault time

Wilson [Wilson 1990] describes an approach that employs both pointer swizzling and page faulting techniques. The basic strategy is to fetch pages of data into virtual memory rather than individual objects. As pages are fetched, they are scanned and all (persistent) pointers are translated into valid virtual memory addresses. References to non-resident objects cause virtual memory to be allocated; these pages are fetched only if the pointers into them are dereferenced. In Wilson's scheme, pages of data in virtual memory only contain valid virtual memory addresses, never persistent identifiers.

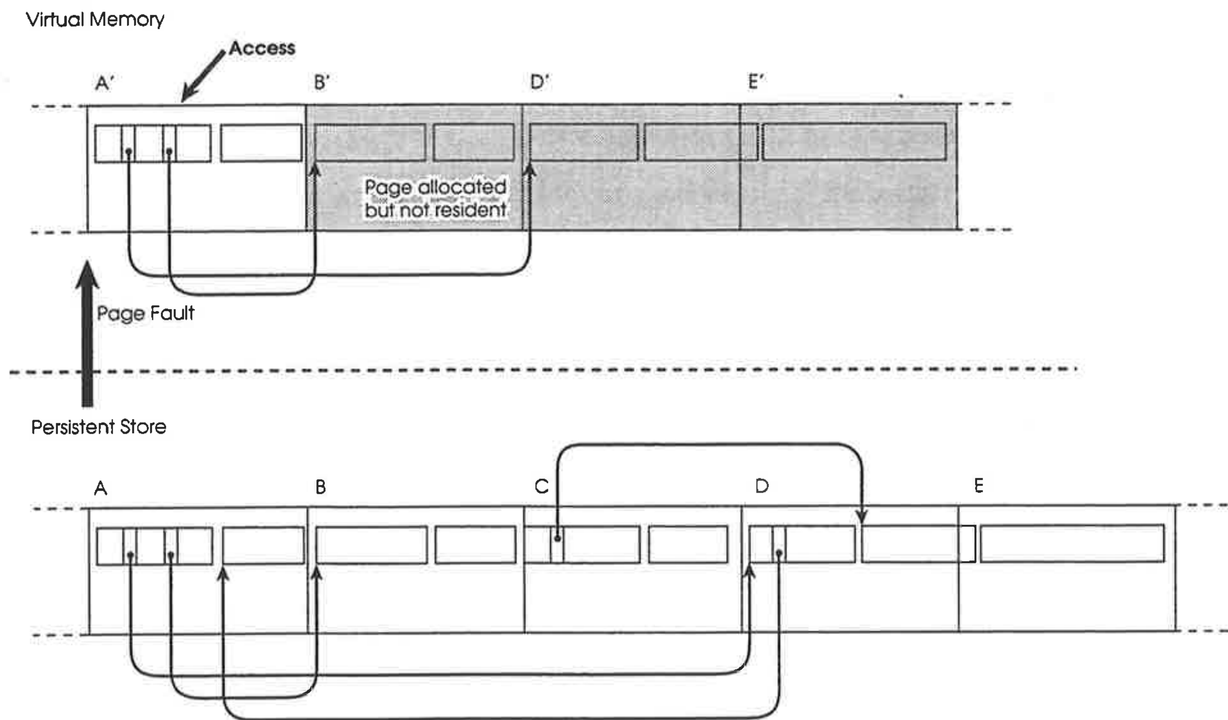


Figure 18. Page faulting and allocation in Wilson's scheme

Figure 18 shows Wilson's scheme in operation; in the diagram, a non-resident persistent object on page A (i.e. an object on a page that has not been fetched into virtual memory) has been accessed. This will cause a copy of page A, denoted A', to be fetched into virtual memory. At this time, the page is scanned and all the pointers in it are swizzled into valid virtual memory addresses. Since page A contains references to objects on pages B and D, locations for pages B' and D' must be allocated in virtual memory and the pointers into those pages swizzled to the addresses of B' and D' with appropriate offsets added. Virtual memory must also be allocated for page E since objects from page D overlap that page. Note that the loading and swizzling of pages B', D' and E' is performed lazily: only space is allocated for them in virtual memory. This mechanism causes virtual memory which may never be used to be allocated. Since pages B, D and E may have already been faulted into virtual memory, a translation table similar to the CPOMS PIDLAM must be maintained to avoid loss of referential integrity.

When a reference to a previously unseen page is encountered whilst scanning an incoming page, three actions are required. Firstly a new translation table entry for the page is allocated. Secondly, the store is interrogated to discover the page's crossing map (described below). Thirdly, virtual memory space is allocated for the page. Interrogation of the store is potentially expensive and since it is performed eagerly, at page fault time, is a potential

performance bottleneck. A later version of Texas has eliminated the second of these steps. It no longer uses a crossing map but instead ensures that page boundaries coincide with page boundaries using padding to fill any resultant holes.

When a page is scanned, it is necessary to find all the pointers on that page. A single bit crossing map mechanim is used to enable objects to be found when scanning a page.

In Wilson's scheme, page evacuation from virtual memory is convoluted. This problem is exacerbated by the fact that virtual memory is eagerly allocated and hence the need to reuse virtual memory addresses potentially more frequent. If a set of pages is written back to persistent storage, the pointers in those pages must be deswizzled into PIDs by consulting the translation table. However, if virtual memory is exhausted and a virtual memory range is to be reused by another persistent page, all pointers which refer to the old contents must be removed.

A translation table that contains an entry for each instance of a referend object can become very large. Wilson proposes a scheme in which the translation table provides a per page rather than per object mapping. To implement this, PIDs are structured so that the offset within the holding page of an object is encoded into the object's PID. For example, assuming 8k byte pages and word alignment of objects, eleven bits are needed to describe the offset. This leaves 53 bits of a 64 bit PID to identify the page. The structure of PIDs is depicted in Figure 20 below.

This scheme has two advantages. First, it is only necessary to maintain a mapping from pages within the large persistent address space to pages in the machine virtual address space. This table is relatively small and of fixed size. Secondly, an object's offset is required in the construction of a swizzled pointer. If the offset were not coded into the PID, further interrogation of the store manager would be required, adding extra cost to the swizzling process.

3.4. A hybrid approach

The CPOMS and systems like it require software tests prior to each object dereference to check if the pointer being dereferenced is a persistent identifier. Wilson suggests that pointer swizzling may be performed at page fault time. This implements a barrier that ensures that a running program may never encounter a PID. However this is not achieved without cost; space must be allocated in virtual memory for every page referred to by data resident in

virtual memory. Whilst this does not seem too onerous it has some unfortunate consequences.

Firstly, space in virtual memory is allocated greedily, this may cause virtual memory to become exhausted even although much of it has not been used. The counter argument says that many programs will have a high degree of locality of reference. However consider an array of large objects such as images – whenever the array is faulted into memory, enough virtual memory must be allocated for all the referenced images. It is likely that such an operation would be common in persistent applications although uncommon in traditional database applications. Applications involving geographical information systems or multimedia systems will suffer especially.

We now describe a hybrid architecture which does not require software checks for pointer validity and does not involve greedy allocation of virtual memory. The architecture is designed to support PIDs which address a space much larger than virtual memory and makes the requirement that PIDs are at least twice as large as virtual memory addresses. From this point on, to ease discussion, we will assume that a PID is 64 bits and virtual memory pointers are 32 bits.

In this architecture, pointers are swizzled in a two phase process: first at page load time to refer to an entry in a translation table and secondly to a virtual address when the referend object is first accessed. When pages are first accessed, they are copied from persistent memory into the virtual address space and scanned to find the pointers contained in them. Rather than allocating virtual memory for every page referenced by the page being faulted in, as happens in Wilson's scheme, the long pointers contained in the page are swizzled to refer to either:

- entries in a translation table if the referend object is not present in virtual memory (*partially swizzled*), or
- a virtual memory pointer (*fully swizzled*) if it is.

The translation table used in this scheme may be similar to either the one used by the CPOMS (a per object translation table) or by Wilson (a per page table). The table contains the persistent and virtual address (if any) of all objects (or pages) referred to by objects resident in virtual memory. For the remaining discussion we will assume a per page translation table. Unlike the CPOMS, the table is protected from any access by the user process, thus when a partially swizzled pointer is dereferenced an access fault occurs. This

triggers the second phase of the swizzle in which the pointer (currently containing the table entry address) is overwritten with the virtual address of the referend.

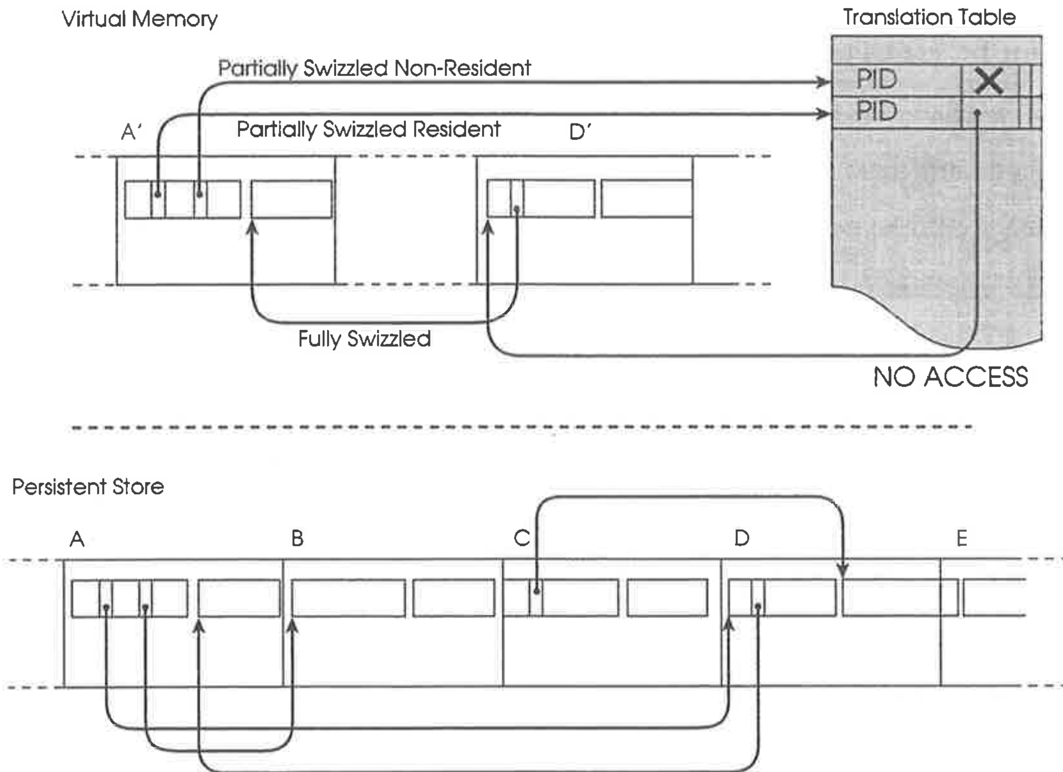


Figure 19. Partially and fully swizzled pointers

3.4.1. Exceptions

Using access protection of the translation table to trigger swizzling and page load has the disadvantage that whilst the mutator must be denied access, the exception handler must be free to read and modify the table. This situation is also found in some garbage collection schemes [Ellis, Li et al. 1988]and the solutions are the same. If the exception and fault handlers are implemented within the kernel they can make use of the full access accorded the kernel to user address spaces. A simple implementation has the exception handler make appropriate protection calls during its execution. These calls can add significantly to the cost of the scheme and make it impossible to allow more than one mutator to run concurrently. Alternatively it is possible to place the translation table within the user's virtual address space but to have a protected area of the same size at high memory to which all the partially swizzled pointers refer. When interpreting pointer values during swizzling and deswizzling the offset between the translation table and the protected area is subtracted from the pointers

to provide the actual address within the translation table. This allows the system to be implemented without modifying the operating system kernel.

A different mechanism for trapping partially swizzled pointer access is available on many newer architectures. This is the unaligned access fault. Many modern processor implementations do not allow for access to data on arbitrary address boundaries [Sites 1993]. This saves complexity in the memory controllers and significantly speeds up the speed of the systems. If an access is attempted to a data item at an unaligned address an exception is generated. Thus legal pointers in such systems may not have their least significant bit set.

If a pointer is stored within the system referring to a translation table entry, but with the least significant bit set, any attempt to access using the pointer will result in an exception. This is true even when access is made to fields within the object referred to by the pointer since the low order bit will propagate through the address arithmetic. This scheme has two advantages.

1. There is no need to protect the translation table from access, or to provide a protected surrogate.
2. The cost of the exceptions are lower. As described in the previous chapter, exceptions involving memory access protection require extra effort on the part of the kernel to discover the precise nature of the exception before it is delivered to the user level.

Thus the translation table may be placed in unprotected memory, and each partially swizzled pointer contains the address of the appropriate table entry but has its least significant bit set. When a partially swizzled pointer is dereferenced an unaligned access fault will result. After masking out the least significant bit of the faulting address the table address can be determined.

3.4.2. Data Load.

If the referend is not resident in virtual memory, the page containing it must be loaded from the persistent store. To do this, the PID, which may be found in the translation table, must be presented to the store manager. Using this the store manager can supply the appropriate page containing the object or portion thereof. Once the page is loaded the partially swizzled pointer is overwritten with the virtual address of the object and the object dereference can proceed. The page load may result in new entries being created in the translation table. In

contrast to Wilson's scheme it is only when an object is used that the store is interrogated to discover how much virtual memory must be allocated.

When a persistent pointer is fully swizzled half the space in the pointer is unused – this space may be used to store the address of the corresponding translation table entry. This allows the pointer to be easily deswizzled. In a partially swizzled pointer the space is used to store the offset within the page at which the object begins. This offset, when combined with the address at which the page is placed when it is faulted into virtual memory, forms the object address of a fully swizzled pointer. The store formats for pointers and the translation table entries are shown in Figure 20.

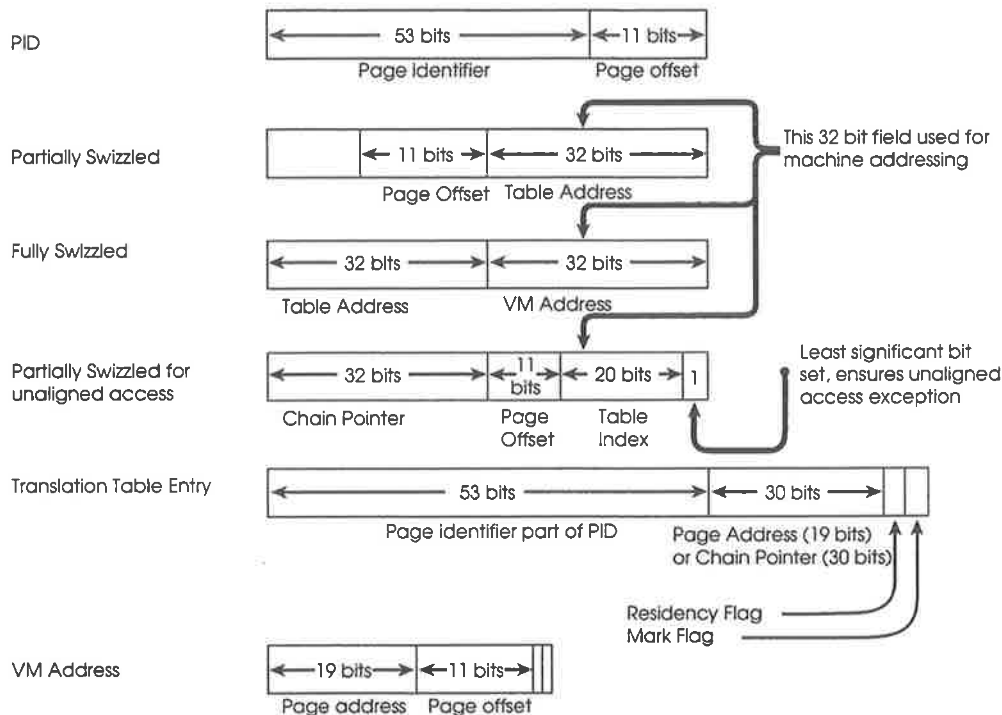


Figure 20 Pointer and translation table formats

The translation table maps from page identifiers in the persistent store to pages within the machines virtual address space. Each translation table entry holds the page identifier field of a persistent identifier, a virtual memory address, a residency bit and a mark bit. If the residency bit is set the virtual memory address holds the address of the corresponding page in memory, otherwise it may contains the head of a partially swizzled pointer chain which is discussed next. The formats depicted in Figure 20 assume a 32 bit virtual address space and a page size of eight kilobytes.

3.4.3. Eager Swizzling

The eager swizzling technique described by Wilson has the advantage that when a page is faulted into memory all the pointers which refer to objects on that page are automatically

correct (since those pointers already refer to the correct virtual addresses on that page). A late swizzling scheme does not have this advantage, however this may be simulated. A form of eager swizzling can be provided by threading a linked list called the *partially swizzled pointer chain* through of all instances of pointers referencing objects on a page. When an object is faulted into memory the swizzling code not only swizzles the pointer that caused the fault, but follows the chain and swizzles as many other pointers as it can. This is eager pointer swizzling; as discussed earlier, this is only more efficient if some of these pointers are used. This very much depends on the nature of the system, programs and programming languages being used and the marginal costs of creating and following the pointer chains versus the cost of on demand per pointer swizzling.

As described the pointer formats do not provide space for the link field needed to implement the partially swizzled pointer chain. The chain may be implemented by using one of the following:

- Making PIDs large enough to accommodate the link. Expanding PIDs to 96 bits also has the advantage of providing a much larger address space. It has the disadvantage of increasing object size.
- Using a per object translation table. Using this technique the translation table pointer field in a partially swizzled pointer uniquely describes the referend object. The upper half of the pointer does not contain the page offset and is free to hold the link field. However per object translation tables can become very large.
- By encoding the information. The problem is that 30 bits are required to implement the chain (assuming word alignment.) The table address requires 28 bits (assuming 16 byte table entries), the offset requires 11 bits, leaving only 25 bits free. Therefore another five bits are required. These bits may be stolen from the table address if the translation table is made 32 times as large as normally required.

  If implemented upon a system using unaligned access faults rather than protection faults the encoding can be achieved without stealing bits. Since dereferencing a partially swizzled pointer with the least significant bit set will always result in an exception the pointer can hold whatever is needed in the remaining bits. To save bits we hold the index into the translation table rather than the absolute address of the entry. Thus only enough bits are needed to hold the index, plus one to ensure the address is unaligned. To cover a 32 bit address space of 8 kilobyte pages this

requires 20 bits, leaving twelve bits to store the within page offset. This leaves the upper word of the pointer free to store the partially swizzled pointer chain. The number of bits needed to store the within page offset increases as page size increases. However the size of the translation table falls with increased page size at the same rate. Thus this mechanism works for all page sizes.

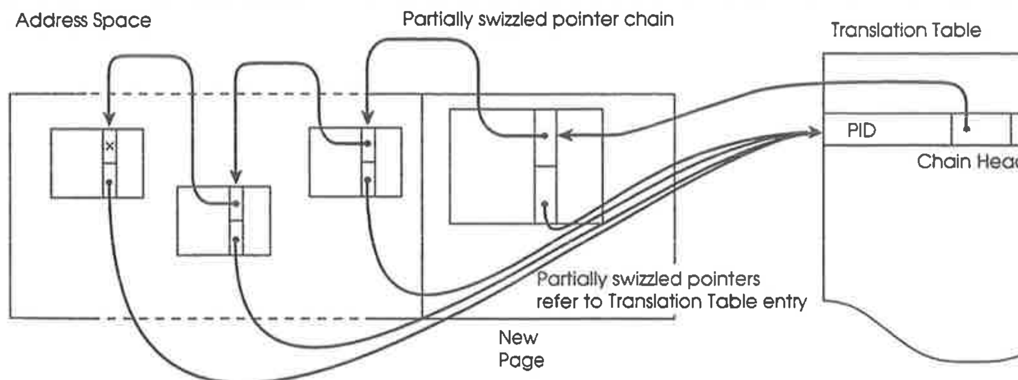


Figure 21 A pointer is inserted into the partially swizzled pointer chain

The partially swizzled pointer chain is formed as pages are loaded into virtual memory. If an instance of a PID is encountered which is already in the translation table, the head of the partially swizzled pointer chain is loaded into the unused space in the partially swizzled pointer and the address of the new instance is copied into the chain pointer head stored in the translation table entry. This process is shown in Figure 21 above.

During the execution of a program, some of the pointers in the partially swizzled pointer chain may have been overwritten by the user making (64 bit) pointer assignments. Such a break is simple to detect when the chain is being scanned since an overwritten pointer will not refer to the expected table entry. If the chains are broken, it is not possible to find all the instances of a partially swizzled pointer. However, the remains of the chain will continue to exist and many of the pointers in it may be still be swizzled through the partial chains referenced by the translation table entry and the pointer being swizzled. Also, future dereferences of pointers in a partial chain will permit yet more pointers to be found and swizzled at low cost. It is possible to maintain intact pointer chains by requiring that code doing pointer assignments perform list insertion and deletion as part of the assignment process. We consider that this would be too expensive for the marginal gains.

3.4.4. Deswizzling

Virtual memory addresses may only be interpreted inside the address space in which they were created. Therefore the only meaningful addresses that can be used in pages outside of a

virtual address space are PIDs. The necessity to make copies of pages outside of a virtual address space arises for two reasons:

- to send pages to a process resident within another virtual address space,
- to send pages back to the persistent store.

This requires the pointers within the page copies to be fully deswizzled (PIDs). This is performed by following the reference to the translation table entry contained within the pointer and overwriting the pointer with the PID found in the table.

The management of pages within the virtual address space involves the allocation and control of two resources:

- physical memory, and
- virtual memory.

Physical memory is a finite resource and is often not large enough to hold the working set of pages used by a program. Pages will be removed from physical memory either to make room for another page needed for computation to continue, or when data is shared between separate virtual address spaces. When a page is removed from physical memory, pointers within it must be deswizzled as described above. A page which is not resident in physical memory may still reside within the virtual address space of the process.

In a persistent system the integration of swap space and persistent storage provides considerable advantages. We therefore assume that pages removed from physical memory are either returned to the persistent store or to another persistent application using a coherent DSM protocal such as that described in the next chapter.

Virtual memory is also a finite resource. Programs that use very large data sets or those which are very long lived may eventually exhaust virtual memory. Indeed, the architecture described in this chapter is designed to support such programs. When virtual memory is exhausted, virtual address ranges require reuse in a manner analogous to the reuse of physical memory. It should be noted that both Wilson's scheme and the hybrid design require that virtual memory addresses be reallocated in such a way that the reallocated ranges do not divide objects.

When a page is removed from the virtual address space, it must also be removed from physical memory if resident. At this time all references to that page from within virtual memory must also be removed. This involves ensuring that all references to objects in the

removed page are partially swizzled pointers by deswizzling the appropriate fully swizzled pointers.

3.4.4.1. Deswizzling in Wilson's Scheme

Wilson proposes a scheme to reclaim pages of virtual memory that works as follows. First all of virtual memory is protected from access. Whenever the mutator attempts to access a page that is protected from access two actions are taken. First, the page protection is removed. Next, the page is scanned to find all pointers on it and any referenced pages are marked. Finally the mutator is resumed. As the mutator executes it constructs a new working set of pages. At some time in the future any page that is neither open for access nor marked as referenced may be reused. Once page reuse has begun it is possible that when a protected page is scanned a pointer to a reused page will be encountered. When this occurs a new range of virtual addresses must be allocated and the pointer changed to refer to this new location. This process is similar to the greedy allocation that occurs when a page is retrieved from the persistent store described earlier.

3.4.4.2. Deswizzling in the Hybrid Scheme

The hybrid scheme provides greater flexibility in address space reuse. Since partially swizzled pointers do not directly reference virtual addresses, fully swizzled pointers may be replaced with partially swizzled. This allows address ranges within virtual memory to be reused whilst references to objects that once resided within those addresses remain in virtual memory. In the hybrid scheme page reuse occurs as follows.

During normal execution a candidate set of page ranges can be identified for reuse, using conventional LRU techniques. This may be integrated with the LRU scan used to manage allocation and reuse of physical memory. When it becomes necessary to reuse virtual address ranges, access to virtual memory is denied as in Wilson's scheme. However, in the hybrid scheme reuse can proceed immediately. Those address ranges considered as candidates for reuse may be reused as soon as their contents are secure in the stable store. An exception will occur on the first access to a page since reuse started, again the exception handler scans the page in the same manner as Wilson's scheme. However rather than allocating new address ranges for those pointers that reference reused addresses, pointers to objects within reused address ranges may be replaced with their partially swizzled form. Thus partially swizzled pointers serve two purposes: to permit virtual memory to be deallocated at low cost and as a mechanism to avoid greedy allocation of virtual memory.

In addition to the mutator causing pointers on pages to be deswizzled, it is advantageous to provide a parallel sweep of virtual memory that eagerly scans pages and deswizzles pointers. Once all virtual memory has been swept, all allocated pages will be open for access and no direct references to deallocated pages will exist. The mutator can attempt to reference a page that is tagged for reuse by dereferencing through a partially swizzled pointer. If this page has not been reused and is still resident in memory it need only be removed from the reuse set and scanned for pointers. The partially swizzled pointer is fully swizzled and execution continues. It is not necessary to reuse all address ranges tagged for reuse. At any time ranges can be removed from the reuse set and references to objects within them left intact.

The ability to choose the number of pages to be reused ahead of time, which pointers to deswizzle, and the rate of progress of the parallel sweep provide useful tuning parameters to the memory management system. Setting the system to label all pages as reused, and to untag any referenced pages upon page scan effectively reduces to Wilson's scheme. Labelling all pages as reused, and deswizzling all pointers encountered effectively frees the entire virtual address space. A complete spectrum of choices is available within these extremes.

3.4.5. Elaboration of detail

The above description glosses over a large number of important details namely:

- finding object addresses,
- pointer comparisons,
- large objects,
- management of the translation table,
- creation of new objects,
- exception handlers, and
- access to the translation table.

We will now proceed to describe these implementation details.

3.4.5.1. Finding object addresses

When an access is attempted through a partially swizzled pointer three actions are required:

1. find the object to which access is being attempted,
2. overwrite the pointer with the virtual address of the referend, and finally,

3. update the saved state of the executing code's register set to refer to the object.

None of these activities is straightforward, and requires detailed study at the basic level of the machine's operation. Consider the code fragment shown in Figure 22 below, a type *tuple* is declared to be a record and an instance of that type is created. Later in the program a field of an instance of type *tuple* is dereferenced.

type tuple **is record**(a,b,c,e,f,g : **integer**)
let an_instance := tuple(1,2,3,4,5,6)
.....
write an_instance.f

Figure 22 Dereferencing a field of a record.

Consider the implementation of the program above. The pointer denoted by *an_instance* may be partially or fully swizzled; an aim of the architecture is to avoid user code having to test which of these it is. Fully swizzled pointers do not present a problem: the dereference is performed without incident. A partially swizzled pointer will result in an attempt to access an address within the translation table and this will cause an access fault. However, the address that causes the fault will not be the address of *an_instance*'s translation table entry since an offset will have been added to the object pointer in order to extract the field. Hence, although an access fault will deliver the address of the fault to the exception handler, the address will not directly resolve the identity of the required object. Similar problems occur in the other two phases; the swizzling code must be able to find and swizzle the object pointer, but ordinarily there is no record of the location of that pointer. If this swizzle is not performed, the system reduces to a translation per dereference design.

In the hybrid system, the saved state of the executing thread is repaired by an exception handler which must therefore be able to determine which machine registers contain the addresses requiring change. This can be arbitrarily difficult; to make the problem tractable steps must be taken to ensure that when an object reference is made, it must be performed in such a way that allows the recovery of the information needed to complete the swizzle. This requires a specification of the object access process at the machine code level.

All of the information required will ordinarily pass through the processor during the execution of a dereference sequence. The difficulty is in keeping track of this information and making it available to the exception handler. A similar sequence is executed whether the access is a read or a write. In general a dereference takes place in three steps and is shown in Figure 23 below:

1. The address of the pointer to the head of the object being referenced is loaded into a register.
2. Using that address, the address of the object is loaded into a register.
3. The offset within the object is added to the object address and the result used as the address of the memory access.

For the mechanism described in this chapter to work, the only changes required to this sequence are to ensure that the pointer address is not overwritten after the pointer value is loaded (which ordinarily is a legal optimisation) and to ensure that the instruction sequence always uses the same registers for this purpose, allowing the exception handler to find the necessary addresses.

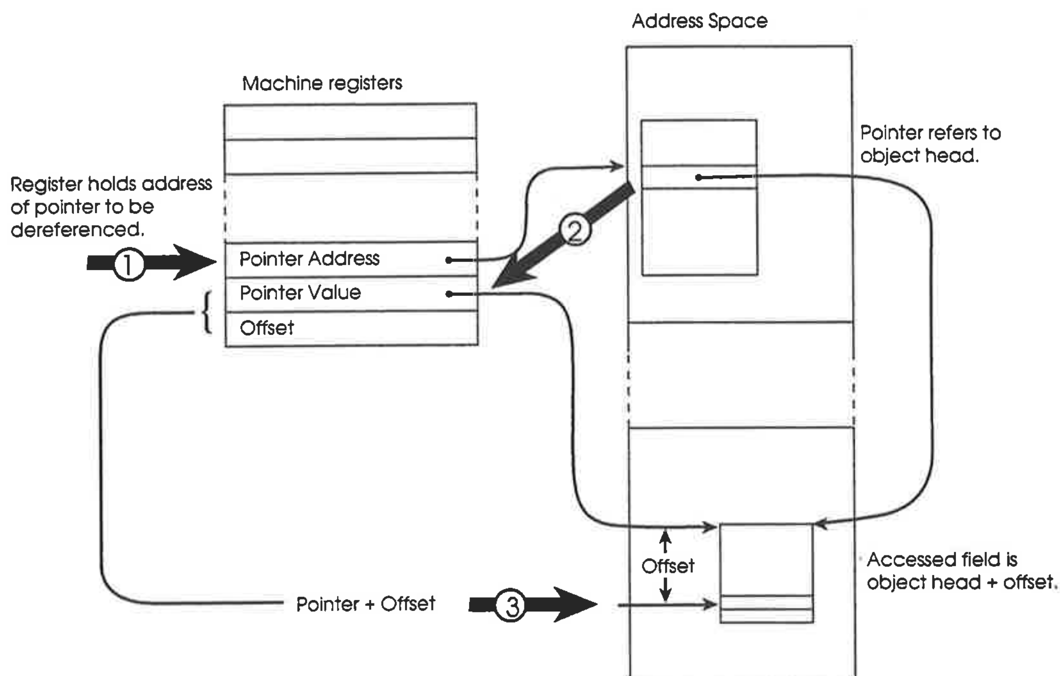


Figure 23. The three steps in pointer dereference

The result of these restrictions is a scheme in which during a dereference operation two registers are reserved for particular purposes. Firstly a *Pointer Pointer* register is loaded with the address of the pointer to the object being dereferenced. Next the *Object Pointer* register is loaded with the value referenced by the Pointer Pointer register. This value is either the address of the head of the object (for fully swizzled pointers) or the a reference to a translation table entry (for partially swizzled pointers). Finally, the offset is added to the contents of the Object Pointer register (with a single indexed addressing mode instruction) and the result used as an address to effect the dereference. If the pointer is partially swizzled an exception will occur. The exception handler will receive either an address within the

translation table (when using protection faults) or an unaligned access (when using unaligned access faults), allowing it to distinguish the exception from any others that may occur. In processing the exception the exception handler places the fully swizzled value of the pointer in both the location referred to by the Pointer Pointer register and into the Object Pointer register, then the instruction that caused the exception is restarted. If the pointer is fully swizzled then the instruction will execute without incident and with no extra cost. This process is shown in Figures 24, 25 and 26 below. The Pointer Pointer and Object Pointer registers are only special during the process of a dereference, they are available for general use at other times.

The access protection mediated mechanism relies on the translation table residing in protected memory and an exception being raised when access to that memory is attempted. When the offset is added to the Object Pointer it is possible for a legal memory address to be generated. This may be avoided if the translation table is positioned with a guard area of protected memory above it. This technique results in the need to restrict object size to be no larger than the guard area maximum object size.

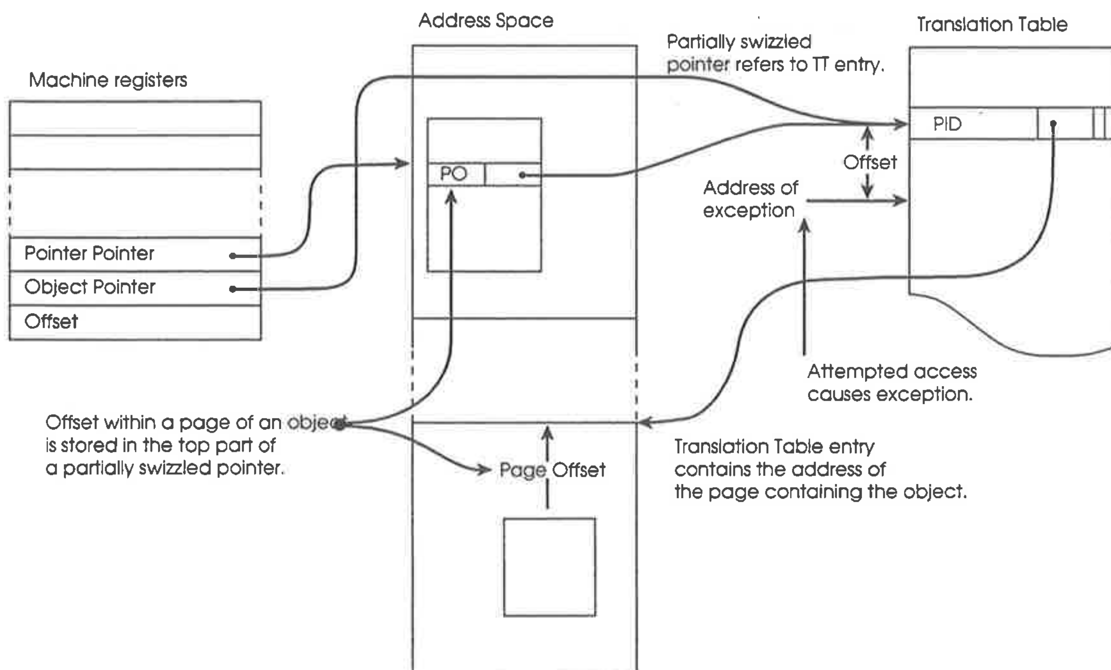


Figure 24. Pointer dereference via a partially swizzled pointer using address protection exceptions.

An implementation utilising unaligned access exceptions does not suffer from this restriction, nor is any guard area required. Instead the value of the pointer is an encoding of the index into the translation table containing the appropriate entry, but with the least significant bit set. The generated code must however never add any offsets to the base pointer except those that

are multiples of the machine word length. This is not an onerous restriction, since under normal circumstances it would result in code that would generate unaligned accesses. Here we are utilising the efforts of the compiler to generate correctly aligned accesses to ensure that accesses are *unaligned*. Some architectures do allow arbitrary alignment of accesses when a single byte quantity is accessed. On these machines the compiler must not generate such code sequences. However such sequences are usually no faster than providing for machine word data movement since the memory controller usually fetches a full word anyway, and provides a read, modify, write cycle for a byte write. Architectures such as the Alpha AXP do not support byte accesses at all and always require the compliler to generate such sequences.

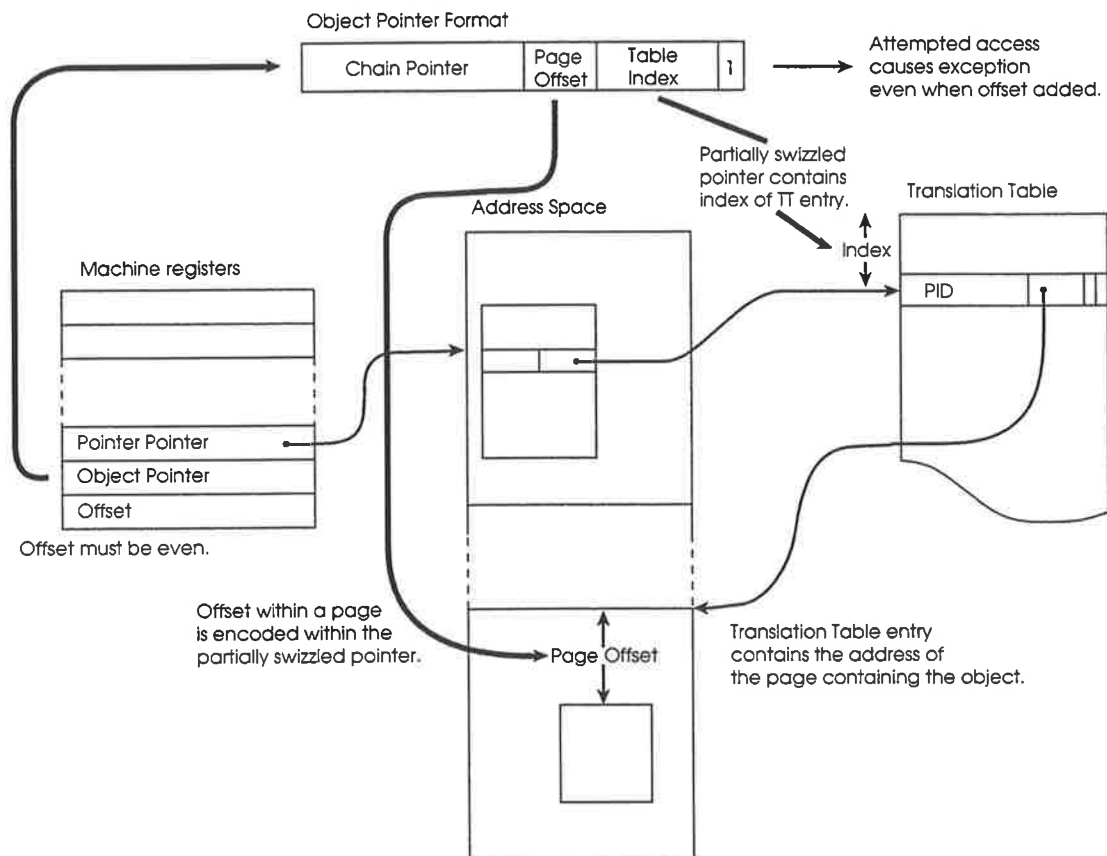


Figure 25. Pointer dereference through a partially swizzled pointer using unaligned access exceptions.

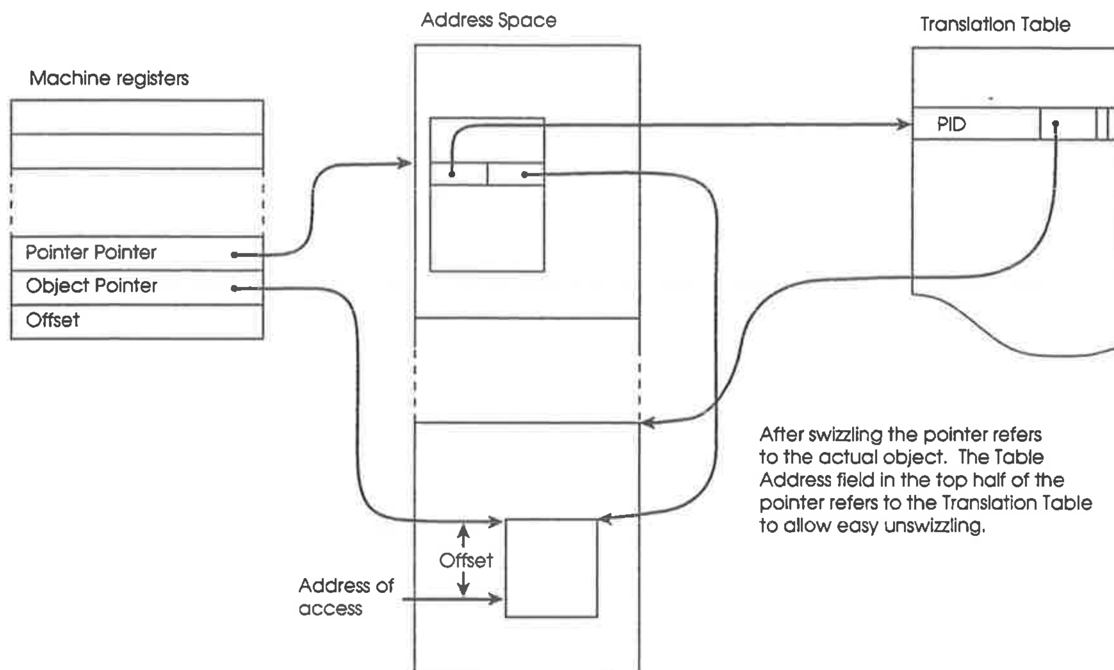


Figure 26. Dereference after the completion of swizzling

The scheme described above is directed at those processor architectures that only support simple addressing modes and require a number of instructions to carry out a dereference. Some processors are capable of executing the sequence described above in a single instruction, on such architectures the exception handler can decode the instruction pointed to by the saved PC. Such a scheme can be more flexible for two reasons. Firstly it may use many different addressing modes and secondly, it is not necessary to designate particular registers since the instruction will indicate unambiguously which registers are being used and for what purpose. However, the complexity of the exception handler is higher. Most architectures with more complex instruction sets allow for arbitary access alignments. Therefore they cannot implement a system based upon unaligned access exceptions.

3.4.5.2. Pointer comparisons

Since a pointer can exist within the system in one of two forms, care must be taken with pointer comparisons. Pointers can either be partially swizzled in which case they contain an offset and a reference to their translation table entry, or fully swizzled, in which case they contain a translation table reference and a pointer to the actual object. These two forms can be differentiated since the translation table and object area occupy distinct address ranges. In both formats a reference to a translation table entry, and the page offset is present, it is therefore enough to compare these values when performing pointer comparison.

3.4.5.3. Large Objects

Objects which cross page boundaries and more importantly very large objects which span a large number of pages require no special treatment. When an object spans more than one page it is not necessary for the whole object to be resident at one time. However it is necessary to reserve enough virtual memory to hold the object in a contiguous span so that it is possible to fault the rest of the object into memory as it is required. This preserves all the advantages that a demand paged virtual memory space has for sparse access to large objects.

3.4.5.4. Management of the translation table

The scheme described uses a translation table similar in format to that used by Wilson. Whereas translation tables in Wilson's scheme are of fixed size, only describing pages in the machine address range, our scheme requires a table that provides entries for every page that is referenced by pointers within virtual memory. Growth of the translation table takes the place of greedy allocation of virtual memory in Wilson's scheme.

The table has two major constraints placed upon its organisation: firstly, translation table entries are referenced directly by objects, therefore table entries may not move. Secondly, the action of swizzling pointers requires that it is possible to find entries from their PID quickly, otherwise the swizzling on page fault becomes a performance bottleneck.

Since pages are removed from the virtual address space, the translation table will eventually contain entries for pages which are not referenced from the virtual address space. By a simple modification to the scan used to deswizzle pointers during reclamation of virtual address ranges, these stale entries can be garbage collected. Any pointers found during the scan may be followed and the mark bit set in the referenced translation table entry. Once the scan has completed, the translation table is scanned and those entries without a mark bit set may be reclaimed. During this scan partially swizzled pointers for which the referend is resident may also be swizzled. Thus the reclamation pass through memory results in all references to resident objects being fully swizzled, stale entries in the translation table being eliminated and the freeing of virtual memory.

3.4.5.5. Creation of new objects

Many objects are created during the execution of user code; many of those objects will be short lived and therefore not require the allocation of a PID. Objects only require a PID when they become visible outside of the virtual address space in which they were created. In practice, this means an object that already has a PID acquires a reference to them.

We now describe a scheme whereby the allocation of PIDs is performed at the latest possible time. Pointers to new objects only contain the object's address; the field that would ordinarily refer to the translation table address is set to a sentinel value that indicates that the object does not yet have a PID allocated. When a page is deswizzled, pointers to objects without PIDs will be detected. At this time, a PID is allocated and a translation table entry created.

3.5. Comparison of the schemes

Table 2 below summarises the main design features and costs of each of the three schemes described.

- *Granularity* is the size of the entity which the swizzling scheme manages.
- *Code compatibility* lists those areas in which specific changes to the code running on the system must be made.
- *Dereference overhead* is the extra cost (if any) of performing a dereference operation.
- *Assignment* is the size of the data assigned in pointer assignment.
- *Object fault overhead* lists the main activities that must be performed when a reference to a non-resident object occurs.
- *Recovery of VM* lists what actions are required when virtual memory is exhausted.
- *Recovery of Translation Table* lists what actions are required when space for the Translation Table is exhausted.
- *VM space allocation* lists the entities for which virtual memory must be allocated.
- *VM space used* lists the entities for which virtual memory is used to hold data.
- *Translation Table allocation* lists the objects for which an entry in the *Translation Table* must be made.
- *Deswizzle action* compares the costs of deswizzling a pointer.
- *Stabilisation Action* lists the actions required to stabilise the state of the system to persistent storage.
- *Large object overhead* compares the use of virtual memory to hold large objects.
- *Sensitivity to exception handler speed* compares how performance is affected by the exception handling mechanism.
- *Overall VM space* compares the use of virtual memory of the systems.

Table 2. Comparison of swizzling schemes.

Feature/System	CPOMS	Wilson	Hybrid
Granularity	Object	Page	Page
Code compatibility	Software check per dereference	No implications	Use of defined sequence for dereference, and pointer comparison
Dereference overhead	Software check per dereference, possible swizzle	None	Usually none, possible swizzle
Assignment	Virtual address	Virtual address	Twice virtual address
Object fault overhead	Copy single object from store	Copy page from store and swizzle internal pointers. For each new referenced page, interrogate store and allocate memory	Copy page from store, swizzle internal pointers and follow pointer chains if used
Recovery of VM space	Rebuild system if VM exhausted	Invalidate VM and rebuild	Invalidate VM and rebuild
Recovery of translation table	Rebuild system if Object Table exhausted	Fixed size table	Garbage collect translation table
VM space allocation	Accessed objects	All referenced pages	Accessed pages
VM space used	Accessed objects	Accessed pages	Accessed pages
Translation table allocation	Accessed objects	Entry per page of VM	Entry per page of VM
Deswizzle action	Follow pointers to PID stored with object	Search translation table for object entry	Follow pointer to translation table
Stabilisation action	Per modified object: Deswizzle pointers, write object to store	Per modified page: Deswizzle pointers, write page to store	Per modified page: Deswizzle pointers, write page to store
Large object overhead	Entire object kept in virtual memory	Accessed pages kept in virtual memory	Accessed pages kept in virtual memory
Sensitivity to exception handler speed	Little impact	Slight impact	High, less when swizzle chain is used
Overall VM space	Lowest	Highest	Low

Each of the three systems described has particular strengths. The CPOMS design is the most parsimonious in the use of virtual memory, but also the one with the highest run time overhead. Wilson's design has the lowest running costs when not page faulting, but the highest page fault costs. If the amount of virtual memory used becomes large Wilson's scheme must incur the cost of rebuilding the working set and expense of an extra translation table. Hence Wilson's design is probably best suited to environments small enough for it never to be necessary to recover allocated virtual memory. Applications with shorter lifetimes and smaller data bases would be most suitable. The hybrid scheme has running costs similar to that of Wilson's design, has lower page fault costs, and is able to recover virtual memory

and translation table space more easily. This is at the cost of forcing the use of a special dereference instruction sequence, and double length pointer assignments.

Pointer swizzling may be characterised by the time at which: pointers to be swizzled are encountered, translation table entries are allocated, memory for the object is allocated, an object is loaded from the store, the initial pointer that refers to the object is swizzled. Further characterisations are: whether other instances of the pointer to the same object are swizzled at the same time, and whether pointers within objects newly faulted into memory are swizzled to refer to resident objects. Each of these activities may be performed either eagerly or lazily, Table 3 below summarises the characteristics of the three systems described.

Feature/System	CPOMS	Wilson	Hybrid
Locate pointers	Lazy	Eager	Eager
Translation Table allocation	Lazy	Eager	Eager
Allocation of VM	Lazy	Eager	Lazy
Object Loading	Lazy	Lazy	Lazy
Swizzle to VM Address	Lazy	Eager	Lazy
Swizzle other pointer instances	Lazy	Eager	Eager/Lazy
Swizzle new pointers	Lazy	Eager	Eager

Table 3. Swizzling system charaterisation.

3.6. Conclusions

This chapter describes three architectures capable of supporting arbitrarily large persistent identifiers and large object stores using conventional hardware. Two of these represent opposite ends of a design spectrum; the third is a hybrid architecture which embodies useful attributes of the other schemes and which has some useful attributes in its own right. The hybrid architecture maintains the advantages of lazy swizzling found in the CPOMS and similar designs namely only allocating space for objects, and fetching objects, when they are referenced. The hybrid design also maintains the advantages of page based designs, requiring no runtime checking of pointers and allowing sparse references to large objects without the need to copy entire objects into virtual memory. A design for machine level dereferencing has been presented that allows exception handling code to swizzle pointers on demand without requiring checking by user code.

Chapter 4. Store Architectures

4.1. Introduction

In the previous chapters we have examined the mechanisms by which implementations of orthogonal persistence manage the contents of volatile addressable memory and abstract over the movement of data between addressable memory and persistent storage. In this chapter we examine the mechanisms through which the persistent store is created and managed.

Persistent systems provide a paradigm in which the programmer is freed from all considerations of the locality of data. The distinction between the volatility of addressable memory and the permanence of disk or other stable storage is removed. Further, a programmer is encouraged to view the persistent environment as immortal, one in which it is both reasonable and natural to place both processes and data.

Conventional operating systems only offer the most basic of mechanisms to maintain such an abstraction. Persistent storage is either completely untyped (such as files under Unix) or rigidly typed (such as in conventional data-base systems). Such systems have considerable difficulty in capturing the full state of a programming system. This is partly due to the semantic mismatch between the data structures used in the supported programming language and the underlying storage mechanisms (consider for instance, attempting to map a program into a relational database), and partly due to the inability of these conventional systems to provide integrity guarantees that allows the programmer to ignore the consequences of system failure (when for example maintaining free form data in a Unix file).

Freed from these failings, and operating within a system in which the entire program state is retained in a reliable manner, a free and rich programming style can evolve. Whilst such are some of the goals of persistent systems, clearly the implementor shoulders considerable extra responsibility in manufacturing such an illusion. This chapter is about manufacturing that illusion and the burdens shouldered.

4.2. Basics

For a persistent store to be truly orthogonal it should provide a store of infinite size, speed and perfect reliability. This frees the user level from any concern over performance, limitations on data and reliability. Clearly these are unattainable aims, however the demands are important and shape the design of persistent stores.

4.2.1. Size

The capacity of a store limits two things: the ability to store new things, and the ability to modify existing things. Clearly if a user must keep track of the capacity of the store, the persistence abstraction is no longer orthogonal. Limitations on modification occur because the store must always be able to capture a new consistent view of the computation without compromising the integrity of the store. This will inevitably require some extra storage to hold the state of modified and new data (whether in separate logs or within the store itself). If only some of the new state can be accommodated, or if the system makes an external commitment that the program state is recoverable (such as might be required in a transaction processing system) then later finds insufficient storage to meet these commitments the system must be regarded as having failed.

4.2.2. Speed.

A system in which the speed of access to some data differs from others in a manner related to the manner of its storage also compromises the orthogonality of the persistence abstraction. It is clearly not possible to manufacture a store of infinite speed. However implementations of persistent stores should add as little penalty in performance as possible and not create any further unevenness in performance. This thesis is partly concerned with attempts to make persistent stores perform as well as possible, and to at least make persistent stores perform no worse than the mechanisms for persistent storage and data access in conventional systems.

4.2.3. Reliability.

The persistence abstraction is compromised if the user of a system is required to explicitly consider issues of resilience and recovery in the face of system failure. Again, perfect reliability is never possible. This chapter only considers issues of recovery from simple machine failure (often termed *fail stop*). Issues of loss of persistent media, replication of data or computation and recovery from incorrect computation are not covered.

4.3. Stability and Resilience

When considering persistent stores two terms are of particular importance; these are *stability* and *resilience*.

- Stability means that data is stored within some persistent media and will continue to reside there. It will survive system shutdown and restart, and will survive cataclysmic failures such as system crashes and power failures.
- Resilience means that the stable data continues to be useful after any such cataclysm. Stability covers the mechanics of storing data, resilience covers the mechanisms used to manage that storage so that some internally self consistent view of the persistent system is always recoverable no matter what calamity befalls it.

When addressing these calamities two separate kinds of failure need to be considered, these are usually termed soft and hard failures respectively.

4.3.1. Soft Failure

Soft failures can be characterised as those that do not result in physical damage to any components. After such a failure the contents of the store are held on stable media.

Causes of soft crashes are essentially these: power failures, benign hardware failures, operating system crashes and user software errors. A study by Tandem [Gray 1990] showed that the vast majority of soft failures were due to software faults (some 80%), the minority to hardware and power failures.

4.3.2. Hard failure

Hard failures are characterised by the loss of, or damage to, portions of the persistent media upon which the system maintains the persistent store. Also included in hard failures is corruption of disk data. Tactics for dealing with hard failures are categorised as:

- Single Unit Failure
- Site Failure

Single unit failure addresses the probability of an individual device becoming defective and failing. Failures include disk head crashes, controller failure, failures which may randomly strike any one device at any time.

RAID (Redundant Array of Inexpensive Disks) is a generic term used to describe techniques where an array of disks is used to provide a single highly reliable logical device. Often the schemes used can also produce logical devices with greatly improved performance. Chen et al [Chen, Lee et al. 1994] characterise RAID strategies into seven categories, these they term level zero through to level six.

Level 0 is a system with no redundancy.

Level 1 is disk mirroring. Fault tolerance is achieved by duplicating each disk drive and ensuring that all disk write operations occur to both disks. Writes may occur asynchronously although some mechanism must be available to detect which disk is the most up-to-date upon restart from a soft failure. Read operations can occur on either device and some performance improvements are available by splitting read requests between disks.

Level 2 is memory style EEC coding. The data is divided between the disk drives by dividing each word of stored data into individual bits and allocating bits to each drive. By supplying extra disks and applying Hamming error correcting coding techniques it is possible to build an array in which the loss of a single disk drive does not result in the loss of any data. Data must be read from all disks or written to all disks for each disk operation.

Level 3 is bit interleaved parity coding. This is an improvement upon level 2 because unlike memory systems, in general, disk systems are able to detect which disk has failed, thus the same level of reliability is assured with only one extra disk. Like level 2 all disks must be accessed in each operation. Simultameous access to all disks results in very large transfer rates but an access rate no better than a single disk.

Level 4 is block interleaved parity. Rather than split data bitwise across disks, data is split into blocks of arbitrary size. For reads smaller than the block-size only one data disk need be accessed. Since data blocks are on individual disks parallel accesses can be performed. Write requests must write the new data and also update the parity disk. A read-modify-write cycle is, in general, needed to update the parity blocks. Thus the parity disk is a bottleneck to performance.

Level 5 is block interleaved distributed parity. This scheme eliminates the parity disk bottleneck of level 4 by distributing the parity blocks across all the disks. Data is also spread across all disks, therefore there is an extra disk across which to spread requests above level 4. Modify requests must still perform a read-modify-write cycle to update the parity blocks but this is now distributed across all disks and ceases to be a bottleneck. Level 5 has become the mechanism of choice or most RAID systems.

Level 6 is termed P + Q redundancy and describes schemes in which Reed Solomon coding [Lin 1970] is used to protect against multiple simultaneous disk failures using a distributed parity mechanism similar to that used in Level 5 designs.

4.3.3. Site Failure

RAID techniques are directed at occasional random device failures. They are unable help where large scale loss or damage occurs. At the other extreme of reliability issues is the loss of an entire site. Loss can occur from catastrophic causes such as fire, flood or terrorist attack or simply the result of power or communications failure that make critical data unavailable.

Applications that require tolerance of the loss an entire sites typically duplicate (mirror) the entire site. Should the primary site hosting the system fail for any reason, operation is intended to resume transparently on a backup site. For the most part these techniques are beyond the scope of this thesis although characterising the replication of data and computation is explored in Chapter 7.

4.4. System reference model

We now introduce a reference model and terminology which will be used for the remainder of this discussion. This model is applicable to the majority of conventional hardware realisations.

Execution must occur through the mutation of data resident within addressable memory. Current realisations of fast addressable memory are volatile, that is they will loose their contents when power is removed. Persistent data in conventional architectures is not directly addressable by the execution engine (processor) and as a consequence a copy of the persistent data must be made in volatile addressable memory to enable persistent data to be manipulated. Managing this movement in a transparent manner was the subject of the previous chapters, managing the state of the store is the subject of this chapter.

To aid in discussion the following terms are introduced.

- Persistent Space: The conceptual environment in which programs execute. Provision of this space is the goal of the systems described.
- Volatile Space: The combination of addressable memory and external state (as viewed by I/O operations) in which programs actually execute.

- Persistent Store: The physical disks and other persistent media which, in combination with their management software, provide storage for the persistent space.
- Persistent State: A particular version of the state of the computation of the persistent space resident and resilient within the persistent store. There may be more than one Persistent State.
- External Guarantee: An assurance that some execution history has occurred and that this history is reflected in the Persistent State. (For example a traditional transaction.)

In a single process, single user, system only one volatile space is encountered and the role of the persistent store manager is to ensure that the contents of the persistent store reflects at least one internally self consistent view of the persistent space, that is at least one persistent state. To create a persistent state the contents of the volatile space must be combined with the current persistent store in such a way as to always yield an internally self consistent new version of the persistent store, one that is in agreement with the external perception of the persistent space. This action is variously termed *commit*, *stabilise* or *meld*. Since the terms commit and stabilise are used in a great many contexts and their use often causes misunderstanding we will follow Dave Munro [Munro 1993] in using the term *meld* henceforth. In its simplest form melding can be modelled as a synchronous copy, directly to the store, of those parts of the volatile space that have been modified since the last meld.

However, the concept of *meld* becomes more complex when external perceptions of the persistent state are considered. Often systems make certain guarantees about the state of the persistent store. In particular, the notion of transactions require that the user level code has some control over the exact time and nature of data melded with the store. As we develop a model of persistent stores this will become increasingly important.

In a concurrent or distributed system there may be many volatile spaces, each reflecting their own view of the persistent space. These views must be coordinated in such a manner that the combination of the component stores is itself internally self consistent. Characterising, modelling and implementing management schemes that provide this guarentee is the ultimate aim of much of the remainder of this thesis.

4.4.1. Self Consistency

Above we used the notion of self consistency in an informal sense. More formally we can describe self consistency in a persistent store as requiring the store to represent the execution of the persistent system either at some particular moment in time or representing some particular correct execution of the persistent space. The latter definition becomes of greater importance as we begin to discuss distributed and concurrent systems where no single execution state is visible at one time. Indeed the state represented within the store may never have existed at any one time. However if it represents a correct possible execution history it is regarded as self consistent. We term this consistency of representation of the persistent state *internal* consistency. We will return to the definition of consistency in Chapter 7 when it will be treated formally in a discussion of distributed systems.

Whilst a realisation of a persistent system is able to control the representation of the persistent space within its stores, often this state only makes sense in combination with outside information, state information over which the store has no direct control. Communication with this external state through appropriate I/O operations must be taken into account by the store mechanisms to ensure that the composite state of external and internal state is consistent, we term such a composite state *externally consistent.* This notion will become of increasing importance as the discussion proceeds.

4.4.2. Synchronous and Asynchronous Creation of Persistent State.

In many systems providing persistent storage, it is the store that provides the user level program with the ability to make external guarantees. For example, using a store interface, a user program can enforce any guarantees required by the higher level views of the system. Often user level requests to meld the volatile state with the persistent state are the only time new persistent states are created. However this need not be the case.

The relationship between external notions of recoverable state and the actual implementation of persistent state is another place where tactics to improve the performance of persistent store are available.

In long lived computational intensive tasks (such as simulation) the creation of new persistent states is useful to allow the computation to be able to recovered from some intermediate state if a system failure interrupts it. Often such action takes place asynchronously to the execution of computation within the persistent space.

4.4.3. Modelling.

We introduce a method of modelling systems which present persistent views of data using four abstractions. These are:

- The external views: the store guarantees the client level that some resilient self consistent states are preserved and available within the store. A store must present at least one such view, some stores may present more.
- The internal views: These are the resilient self consistent states which the store contains. The store has not made any external guarantees about the availability or permanence of these states.
- A baseline: this is a distinguished self consistent persistent state internal to the store which fully captures the state of the supported client system. One of these will always exist within a store, trivially a newly created, empty store is one such baseline.
- A stable but inconsistent state: these are the various pieces of data copied from the persistent space which do not currently form part of any internally consistent state representation.

The store is modelled as a single synchronous entity. Relative to the store a single computation proceeds on a time-line of totally ordered events. The four abstractions above represent state at specific instances of computation on this time-line. These can be represented in what we term and BIES (from Baseline, Internal, External and Stable) diagram.

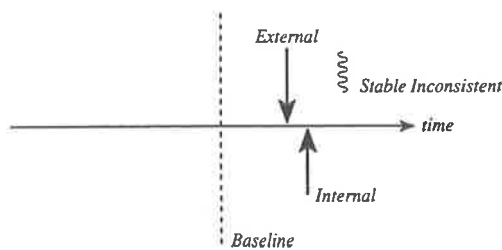


Figure 27. A BIES time line.

The componets of a BEIS diagram are depicted in Figure 27 above. The diagram represents progress in time progressing to the right, individual aspects of the store state are placed on the time line at the times for which they represent the history of the store.

In representing the state of a store with a BIES diagram we can note that any external view must correspond to some internal view, although additional internal states that have

not been associated with an external state may also exist. Stable but inconsistent state will in general represent state information in advance of the external view, any such state preceding the external view is of no use and subject to asynchronous deletion.

4.5. Mechanisms

A store viewed externally presents nothing but its external views, these are guarantees of the availability of resilient states, states to which the store will be able to restore the state of the persistent space. Internally the representation of data may be implemented using a number of tactics. Maintaining the external semantics of a store whilst exploiting the allowable gap between the volatile state and persistent state is the core of store management techniques. This is the subject of much of the remainder of this chapter.

4.5.1. Snapshots

The most basic mechanism in creating a self consistent view the persistent space is to take a copy of the volatile space as an atomic action. This mechanisms has become known as *snapshotting*. In terms of the BIES representations, a new snapshot is a new baseline, and both the external and internal views of the store are tied to the baseline. Once a new baseline is created the old one is obsolete and whatever resources it used can be reclaimed. Stable but inconsistent state only exists whilst the new snapshot is created, the mechanisms of creation and the role of stable inconsistent data will be discussed later in this chapter.

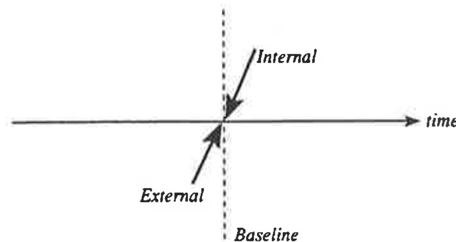


Figure 28. Snapshot time-line.

Snapshots are a simple and valuable mechanism and are employed in the majority of persistent store implementations. Since a snapshot captures a complete view of the system's state without requiring any extra work to re-create this state when needed, snapshotting schemes are often termed *no-undo no-redo* [Bernstein 1987].

4.5.2. Incremental baseline generation

In systems in which data is asynchronously written to the store (in particular systems which integrate swapping of virtual memory with the stable store operation) stable data is

recorded which is inconsistent with the other data in the store. This data can be utilised in the generation of new baseline states.

In Figure 29 below three data items (A, B & C) have been modified by the user level program. These items are asynchronously copied back to the store where they are stored in such way as their previous versions (those that form part of the baseline) are still retained. When at some later stage all three data items have versions resident in the store that are consistent with one another a new baseline can be created.

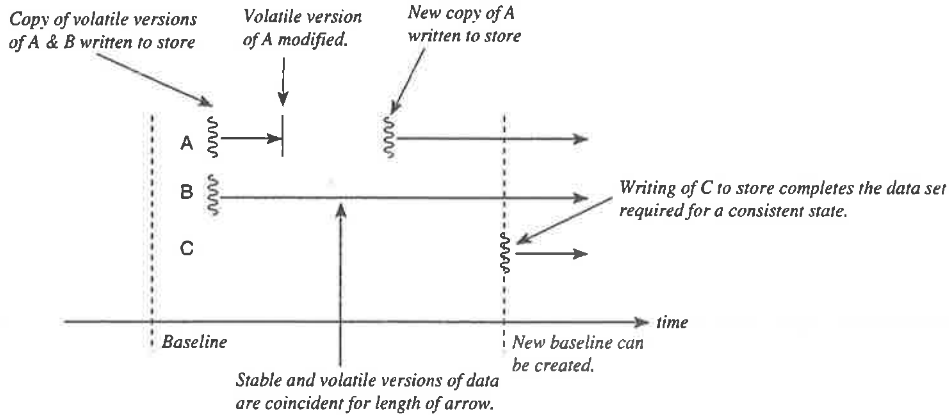


Figure 29. Asynchronous copies of data can be assembled to create a new baseline.

4.5.3. Logging.

The state of the persistent space need not be represented by a snapshot. Once one snapshot exists within the store (the baseline) other versions of the persistent space can be represented by a sequence of changes, commonly termed a log. Logs allow the external view of a store to be derived from a baseline snapshot made at a different time to the external view. Thus the persistent space is reflected in the store by a combination of baseline and log. An important part of this notion is that the baseline snapshot may be in *advance* of the guaranteed external view. In this case the log maintains a set of changes which effectively undo computation and allow the store to present the required external view upon recovery.

In order to make appropriate guarantees of recoverable state, the log must itself be resident on, and recoverable from, persistent media. The balance between the frequency of creating baseline snapshots, size of logged information and the frequency of writing logs to the store allows tuning of implementations to best serve the demands made upon the store.

4.5.3.1. Modelling Logging.

Logging may be divided into two different mechanisms; both involve the recording of changes to some baseline. Logs that describe changes to a baseline that existed earlier are

known as a *redo* logs. Logs that record changes relative to a baseline that represents a later time than the times represented by the individual log entries are known as *undo* logs. Both kinds of log can be combined together allowing an external state to be regenerated either side of a baseline.

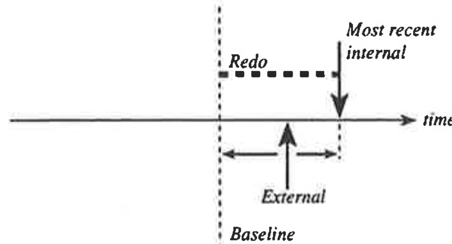


Figure 30. Redo log. The external view may range from the time of the baseline to the end of the log.

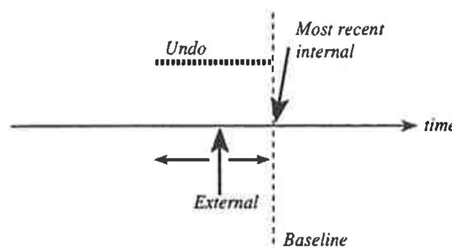


Figure 31. Undo Log. The external view may range from the time of the beginning of the log to the time of creation of the baseline.

Logging can potentially provide extremely fine grained recovery. Each log entry may hold arbitrarily small changes to the state of the store. However, in order to use such fine grained recovery, log entries reflecting these changes must be written to the persistent store with the same period as the granularity in time for which recovery is expected. Usually this period will be coincident with the advancement of the external view of the persistent space. Thus, no matter how fine grained the log entries are, they need not be written until the external view advances. The opportunity therefore exists to batch together log entries and improve performance.

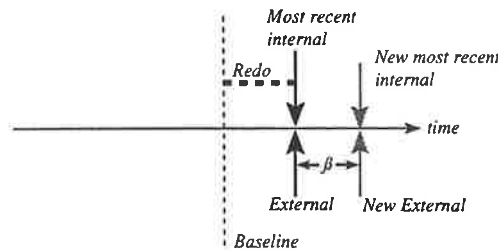


Figure 32. Redo log. Log entries in interval ß can be written in a single operation, often improving system performance.

4.5.3.2. Undo and Redo in the one store.

Clearly a store can implement a combination of both undo and redo logging. Such designs allow the external view to range over times both before and after the baselines creation. Unsurprisingly such tactics are termed *undo-redo*.

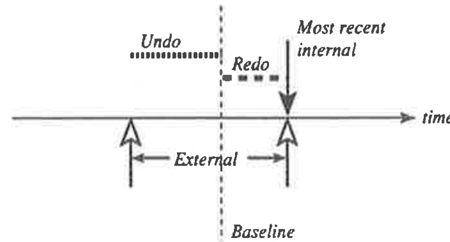


Figure 33. Undo Redo Logs. The external view can range over a large range.

4.5.3.3. Recovering logs

Logs can potentially grow indefinitely. As execution continues some mechanisms are required to recover the space occupied by logs. Such techniques usually involve building a new baseline. Whenever a new baseline is generated any part of a redo log describing changes earlier than the time of the new baseline can be truncated. A new baseline can be created in one of two ways.

1. A new baseline can be generated by snapshotting the current state of the volatile space into the store. This results in a baseline that exactly corresponds with the current state of execution. Care must be taken if this new baseline is in advance of the external view. An undo log must be created from which the external view state can be created from the new baseline, or the old baseline and log must retained until the external view advances past the new baseline.

2. A new baseline may be constructed by applying the changes listed in the redo log to the current baseline. Traversing the log can yield a new baseline at any of the times for which log entries were made and still exist. Such a baseline is thus independent of the current volatile state. Thus a baseline can be constructed that corresponds to the external view without requiring the construction of an undo log.

The first option has become known as *immediate update*, the second as *deferred update*. The terms *immediate* and *deferred* refer to the relative times between the real time and the time represented by the new baseline. Where immediate update is used often the creation of the new baseline is done when the external view of the store is advanced (the users program requests a meld) thus avoiding the need to create an undo log.

4.5.3.4. Replay Logging

So far we have not considered logging beyond a simple notation of whether logs retain either undo or redo changes. However the nature of the information logged has a great bearing on the nature of the store system. At their most basic log entries may contain a byte for byte description of changes that affect the raw data within the persistent store. (Such descriptions are sometimes colloquially known as *diffs*, from the name of the Unix utility that creates difference summaries.) However this is not the only possibility.

Often the higher level programs which operate upon a persistent space do not embody any state of their own. A database engine is a good example. Such systems mutate the persistent space in response to high level commands. If such a system does not include extra state other than that within the persistent space and operates in a deterministic fashion, logging of the external command stream provides exactly the same information as a redo log which logs the changes to the persistent space. This technique is usually termed *replay*. Since replay is usually impossible to construct in such a way that it can run backwards, undo logs are rare.

Figure 34 below, represents such an equivalence. The units depicted respond to a high level command stream. The system on the left represents the persistent store as a utilising baselines and a redo log. The redo log lists incremental changes to the baseline state. The unit on the right logs input commands. In both systems re-creation of an external view involves traversing the log, the system on the right replays logged commands to the compute engine. Both systems are externally identical.

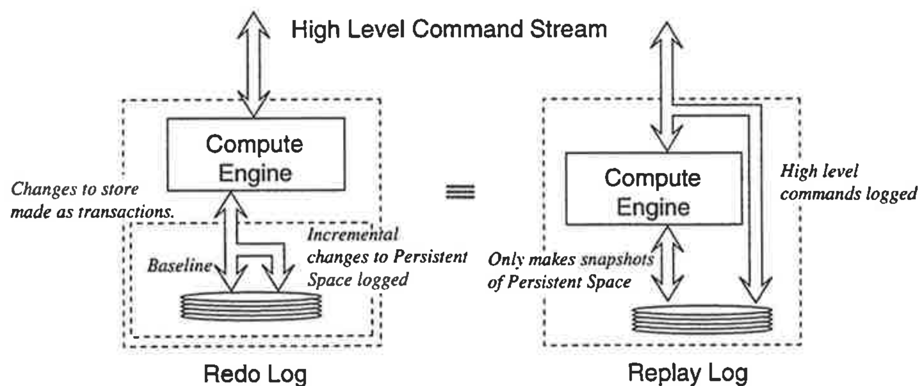


Figure 34. Equivalence between redo and replay logging schemes.

Deterministic computation does however require extra care in implementation. In general, all outside communication must be logged and be capable of re-execution. Such communication must include system calls and any other contact with the environment external to the persistent space. For instance replay of a call to the system time function

must appear to return the time of the original computation, not that of the replay. Secondly, truly concurrent programs are prohibited. They may however be simulated by co-routines in which scheduling is deterministic. Implementation strategies and operation system designs to provide such a computation environment are covered in detail in Chapter 7

4.5.4. Nested Store Strategies

Schemes where high level logging is used can be described in terms of nested BIES models. Figure 35 below, depicts the same scheme as that from Figure 34 above, now revealing the BIES representation. In addition to the redo log used to implement replay, the system depicted also uses a redo log, of changes to the underlying data base. In this example the inner store may log coarse grained changes to the store whilst the external view may log input commands at a much finer grain. The users view of the entire system is one of very fine grained reliability.

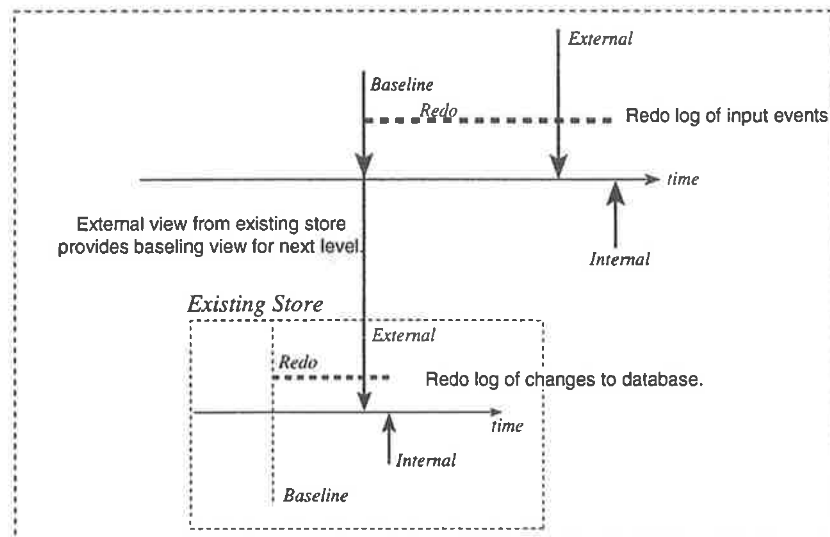


Figure 35. Nesting of store models represents high level logging.

4.5.5. Output in replay systems

When a system reconstructs its state from logged information through replay, output operations will also be replayed. Two options are available for dealing with replayed output.

1. If output operations are idempotent they may be safely replayed. This may involve unnecessary effort during replay, but requires no special action during normal logging.
2. If output operations are not idempotent, or the extra effort involved in output during replay is considered onerous, the system can employ a mechanism to discard output. To do so requires that a stable counter is kept of the number of

output operations performed in the original computation. During replay a count of replayed output operations is kept and all output discarded whilst the replayed output counter is less than the counter for the original computation. This requires the extra effort of maintaining a stable counter of output operations. Update of this counter must be atomic with the output operation.

Unfortunately such update is intrinsically impossible. However careful implementation can make failures during such an atomic output arbitrarily unlikely. Battery backed-up, or other non-volatile addressable storage is often utilised to improve the reliability of such mechanisms. Some peripheral devices are also designed with non-volatile storage and implement a transactional output abstraction, further improving reliability of atomic output.

4.6. Concurent Access, Distributed Systems and Multiple stores.

Thus far we have only considered systems with a single logical store and a single user process. When more complex systems are to be built we are faced with the existence of many concurrent users of a store system, the use of many separately managed stores and a combination of the two. The characterising of such systems and the manner in which they operate is only described in general terms, detailed discussion will be left until later chapters. Here we are interested in the demands these systems place upon the design of the stores themselves and tactics which store implementations can use to aid the operation of such systems.

When faced with many stores we are faced with the possibility of many external views of persistent state, each corresponding to a different time. The manager of the combined store is responsible for coordinating these views so that there always exists at least one self consistent global view. This consistent view is know as a *consistent cut* [Schwarz and Mattern 1992].

The simplest approach to the creation of a consistent cut is for the global store manager to synchronously force each internal store to create a new external view. Thus each store provides an external view at the same time and the global view is trivially comprised of the component stores. However this is both heavy handed and usually unnecessary.

Three important tactics allow the global store manager greater freedom in maintaining a consistent global view. Theses are:

- causality tracking [Schwarz and Mattern 1992],

- replay of output, and
- exploiting log granularity.

4.6.1. Causality Tracking

A component store of a multiple store system can individually meld so long as the modification of data in the store remains independent of the modification of state in other stores. If the global store manager has access to the communication between separate parts of the system it can determine the causal interdependencies of its constituents and is only required to force synchronous melds of interdependent stores.

Stores used in such systems can aid the management of the distributed persistent state by providing more than one external view. This can be accomplished by allowing more than a single baseline to exist in snapshotting stores or by exploiting the nature of logging, discussed next (or indeed a combination of the two).

The distributed Casper system described in Chapter 6 implements a limited form of causality tracking to achieve the opposite effect. By tracking the interdependencies of individual concurrent user programs, a single store performs the minimum work needed to take a snapshot of a subset of user programs at a time. It operates with a single central store that implements a noundo-noredo snapshotting store.

4.6.2. Replay of Output.

Thus far we have described replay logs as requiring stable storage of their own. However if a program is simply dependant upon the output of another program for its input we may be able to dispense with the full contents of a stable relay log by recreating the input stream through replay of the generating program. This places the some of the burden of reliability upon the generator program and the store that holds its state. A simple chain of processing can therefore be recreated without requiring full logging of input at each stage of computation. However more complex patterns are possible. Programs may be mutually interdependent upon one another for the creation of input events, thus potentially creating unresolvable interdependencies. Mechanisms to control such computations are described by Mattern [Mattern 1990], Sistla and Welsh [Sistla and Welch 1989], Strom and Yemini [Strom and Yemini 1985], and Johnson and Zwaenepoel [Johnson and Zwaenepoel 1990].

Stores used by these schemes must provide a logging capability (although truncation of the log is partially under the control of the distributed control mechanisms) and stores utilising the Strom and Yemini algorithms must be able to maintain multiple snapshots.

(Truncation of the history of baselines is also under the control of the management algorithm.)

4.6.3. Exploiting Log Granularity

Logging provides the ability to regenerate the contents of the store at a fine grain. By applying only part of a log upon recovery a store is able to generate a continuum of possible times, one for each log entry. This can considerably ease the constraints for finding a consistent cut in a multiple store system. Logging stores therefore do not simply provide a single external time at which they can provide a self consistent state, but rather provide a set of times.

A system in which multiple persistent stores must be reconciled to maintain a consistent global state can take advantage of this freedom. Such system can avoid synchronised creation of external views when creating a global external view, and allow individual stores managers to execute asynchronously. In Figure 36 below, a single global store presents an external view that is composed of two separate stores. Both internal stores are implemented using redo-logs. By making use of knowledge of the range of external views possible from these stores the global manager can construct a consistent state. The overlapping range of log times allows an external time to be created from a time starting at the beginning of the log of Store B up to the end of the log of Store A.

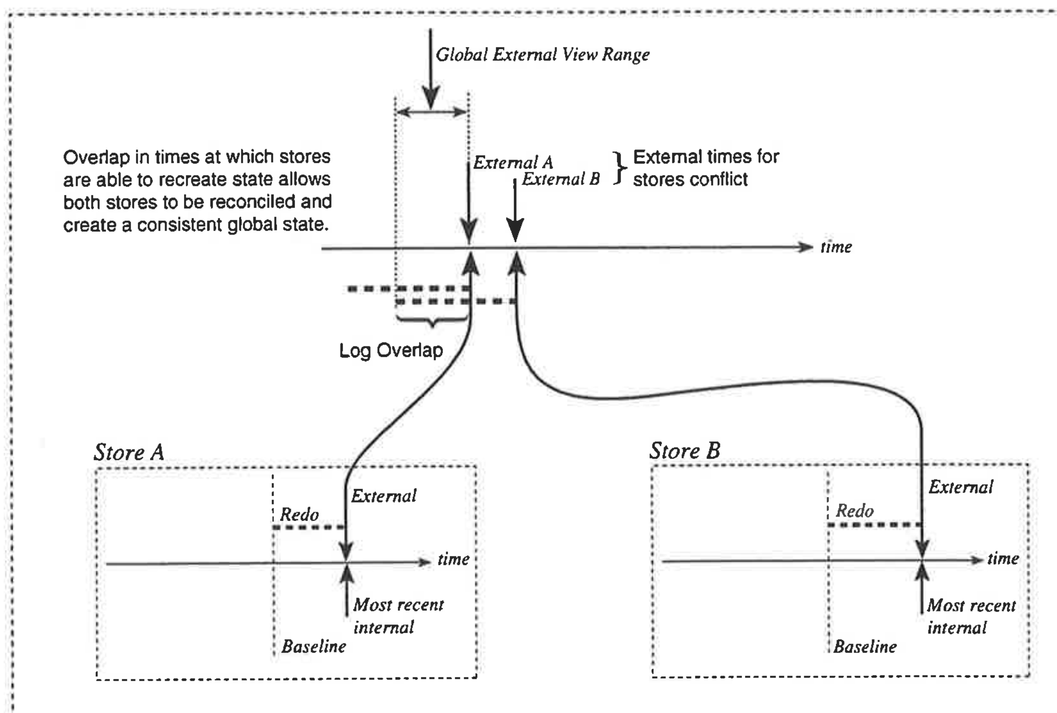


Figure 36. Logging creates greater freedom in reconciling stores.

This form of reconciliation of multiple stores is characteristic of those required by the Johnson and Zwaenepoel algorithm. It allows a single central coordinator to use baselines and logs in individual stores to provide a single global time.

4.7. Trade-offs

Clearly the majority of implementations will aim for the highest possible performance. However in general there is a clear trade-off between three competing store requirements. These are the speed of:

- operation during normal running,
- recovery after a failure, and
- advancement to new external view.

The design aims of the user level system will largely dictate the strategy taken in store implementation. For example a transaction processing system will probably aim for highest performance in generating new external views, and thus may elect for a redo or replay logging scheme. Systems which provide highly available critical systems may elect to place highest priority on speed of recovery and may elect to use a noundo/noredo (for example a snapshotting scheme), where a consistent persistent state is always available. Persistent programming environments may elect to use strategies that emphasise performance in general operation, and schemes which allow for integration with system page swapping.

Logging systems provide the opportunity for creating hybrids of performance goals. For instance aggressive truncation of redo logs can be performed asynchronously and deliver fast transaction processing with very good failure recovery performance.

4.8. Implementation Strategies.

A common denominator in store implementations is that at least one internally self consistent and useable version of the persistent state is always available on stable media. The exact manner in which the state is represented is open to considerable variation, but like physicians obeying the creed “Do no harm” implementors of persistent stores heed a similar maxim. At no time may any situation arise in which the stable state of the system does not contain at least one consistent state. Two important steps are discussed below. These are:

- The mechanism by which a store atomically progresses from one external view to another. The mechanism discussed is Challis's Algorithm.
- The mechanism by which persistent state is held whilst stable inconsistent data is also stored and the manner in which the two are reconciled. The mechanisms presented are generically termed *shadowing* stores, and we discuss two tactics for implementation of shadow stores.

4.8.1. Challis's Algorithm

In all persistent systems there must be a stable fixed point from which the recoverable store state can be found. The creation of this fixed point is complicated by the nature of the hardware commonly used. Almost all persistent storage is implemented atop magnetic disk drives. These have the common attribute that all data is stored in fixed sized sectors, typically of 512 bytes, although some devices support other sizes. Thus the smallest datum that can be written as an atomic operation is a sector in size. However when confronted by all possible failure modes of a system it is not possible to guarantee that a sector write will complete successfully. The problem of creating a new descriptor for a store is solved by a mechanism described by Challis [Challis 1978]. The mechanism operates as follows.

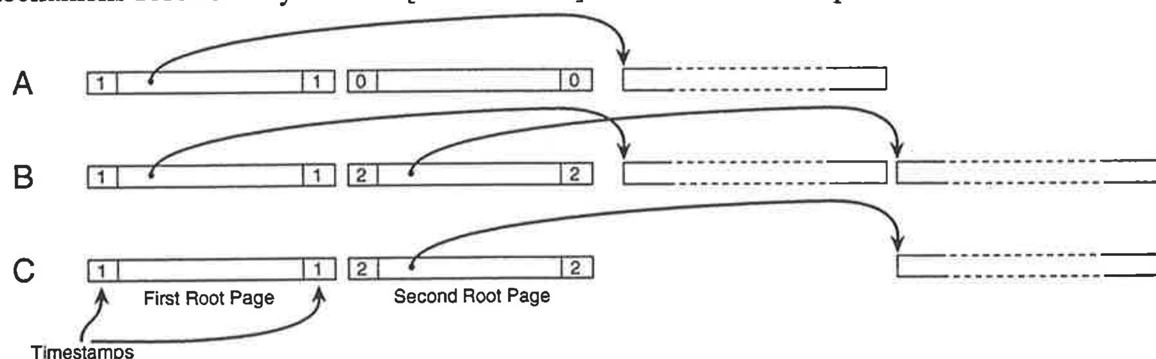


Figure 37. Challis' Algorithm

The store maintains two separate sectors on disk each of which can each describe the state of the store. These are often termed the *root* pages. At any time at least one of the two root pages will describe a stable and self consistent state of the store. In Figure 37 the state at time A shows the first root page describing some store state. When a new version of the store state is created, it is performed in such a way that the current stable state is not damaged. A new state superceeds the previous stable state when new description of the store is written to the alternate root page (the page that does not describe the existing stable state). State B in Figure 37 shows this. At this moment two self consistent versions of the store exist. These states are identified by time-stamps maintained in the root page. The root

page with the most recent time-stamp is considered the current root page, the other is ignored. Once the new root page is successfully written to disk the stable storage used to represent the penultimate state may be reclaimed, as depicted in state C of Figure 37.

To detect if writing the root page to disk failed part way thorough, the time-stamp is written at both the beginning and the end of the root page. Assuming that sectors are written linearly we can assume that if the time-stamp is successfully written to both ends of the sector the intervening data was also successfully written. Upon recovery the most recent root page, with both time stamps in agreement, is chosen as the basis of the store. Even if a failure occurred during the final writing of the new root page the previous stable state will be available. Thus the store is resilient against any single failure of the system during the creation of a new store state.

4.8.2. Shadowing

When the store system is required to move from one self consistent state to another it must do so in such a way that there is never a window of opportunity in which a failure can occur when there is not a self consistent persistent state available after recovery. Challis's algorithm above describes how this can be acheived for the store's meta-data. Here we describe the manner on which this is acheived for the stored data.

Whenever data is written to the store that has been modified since the currently held persistent state was built, it must be placed in the store in such a way that the existing versions of the same data remain available. The mechanisms by which this is acheived [Lorie 1977] are termed *shadow* mechanisms. Shadowing is effected in one of two ways. These are termed *before-look* and *after-look*. Of the two, after look is the easiest to describe and is covered first.

4.8.2.1. After-Look

An after look mechanism writes new versions of data to a new location in the store, leaving the old version alone, and thus any existing persistent states untouched. When the new data is to be melded into a new persistent state (a new baseline) a new version of the store meta data describing the location of data in the store is created. Once this is performed Challis's algorithm is used to atomically move to the new meta data is used. An after look store always has the most recent persistent state available for immediate use after a failure. It is therefore a noredo-noundo mechanism. After a new baseline is established the storage used to contain obsolete data and meta data can be recovered.

Since the after-look mechanism allocates new space for newly modified data, the strategy has the effect of scattering data across the store as modifications occur. Performance gains acheived by clustering data together within the store will therefore be slowly degraded as the store operates.

4.8.2.2. Before-Look

The before-look mechanism writes new versions of data to the location in the store in which the original version is retained, overwriting it. In order to avoid compromising the resiliency of the store it must first ensure that the original data is safely retained elsewhere in a manner that will allow it to be replaced should a failure occur before a new persistent state is created. Writing of new data to the store is performed in three steps.

The old version of the data is copied to a new location in the store (the shadow location). Next a structure describing the location of the shadow entries is written. Finally the new data is written to the location in the store the original occupied.

Should the store fail before a new persistent state is established the recovery mechanism must recognise the existence of the shadow mapping structure, then use this structure to re-establish the old contents of the persistent state. When a new baseline is created the shadow map and shadow data is deallocated.

Clearly a before-look strategy is an undo system. The contents of the shadow space have a strong similarity to an undo log. However whereas a log can be partially applied to yield a system state at an intermediate time, a before look shadow cannot. Once the first shadow is created for a modified datum, subsequent writing of further modifications simply overwrite the datum in place, no extra shadow copies are made. Since data is overwritten in place, any clustering of data within the store is preserved.

4.8.2.3. Shadow Paging

Shadow stores are a natural mechanism for the implementation of page based systems. This is especially true for those systems which integrate system virtual memory swapping into the store mechanism, since the shadow area naturally provides storage for swapped data. In the next chapter a number of implementations of page based shadow paged stores is examined.

Figure 38 below depicts the steps involved in creating both before and after-look shadow entries in page based stores.

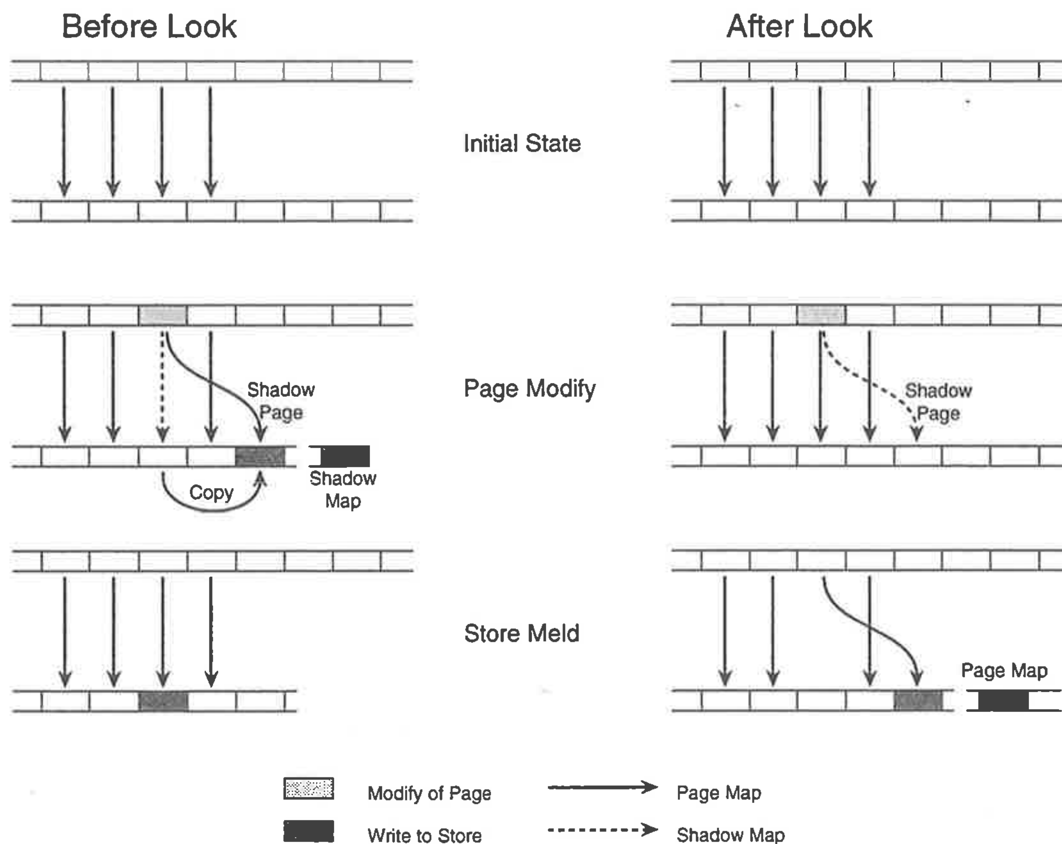


Figure 38. Before and after look page map mechanisms.

4.8.2.5. Maintaining locality

After look schemes can recover much of the on disk locality of data they loose relative to before look schemes. Two simple schemes are illustrative.

Modern disks often maintain a large cache within the controller which holds the entire contents of the just read track. Moreover, access to blocks of data is faster if the next data requested resides in the same cylinder as the last read data, and the heads do not need to move. Allocating data into cylinder groups is a technique used by the Unix Fast File System that forms part of the Berkeley 4.3 release [Leffer, McKusick et al. 1989]. A similar technique can aid performance in after look stores. Such a scheme requires that a predetermined fraction of each cylinder is allocated for shadow use. This allocation will shadow blocks in that cylinder group. As computation proceeds the actual identity of the reserved shadow blocks will change, but the fraction of each cylinder allocated remains the same. Choosing the fraction to pre-allocate is dependant upon the nature of the computation. Clearly an allocation of one half of the cylinder will guarantee that shadow blocks are always available, at the cost of halving the data that can be stored on a disk. Allocating less than half requires that some form of overflow allocation mechanism is also provided.

A second mechanism to improve locality on disk is to use an asynchronous process to move blocks of data. This is analogous to disk de-fragmentation tools used with some operating systems. A stand-alone de-fragmentation tool must operate with the same care as the store manager itself. It must always ensure that a self consistent store is available on disk. Accordingly changes to the disk layout must take effect as atomic operations. A de-fragmentation process can also be integrated into the store manager. Such a de-fragmented may operate by modifying the volatile descriptions of the store. Later, during normal operation, when the volatile store description is melded with the stable store all changes will become persistent, including the results of any de-fragmentation of the store.

4.8.3. Object Shadowing

Instead of shadowing pages of data it is reasonable to shadow full objects from the language level. Such a scheme is used in Argus [Liskov, Curtis et al. 1987]. Shadowing of objects rather than pages introduces a different set of trade-offs.

When shadowing objects the entire object must be copied, this works in favour of small objects, but becomes an increasing burden as objects become larger in size. Disk mechanisms require a fixed size datum (block) to be written with each operation, typically of 512 bytes but sometimes larger. For a before look scheme based upon objects, each object modification will require this minimum transfer to effect the copy of data in the store no matter how small the object. Furthermore the cost of initiating disk operations is fixed (and high), no matter how much data is transferred. An after look scheme may cluster modified objects together onto blocks and write them to the store in a minimum of operations. However, such clustering will destroy any other clustering of objects that may be attempted to improve performance.

4.9. Conclusions

We have described a spectrum of storage strategies and implementation techniques for the provision of resilient stable storage. In the remainder of this thesis we will explore the design and implementation of distributed persistent systems. Each of the store strategies described will be seen to have its own particular role to play in such environments.

Chapter 5. Implementation Strategies

5.1. Introduction

In the previous chapter we have examined mechanisms for exploiting page based hardware in the provision of language mechanisms and abstraction over data movement in persistent systems. In this chapter we examine implementation strategies for page based stores.

5.2. Related Work.

5.2.1. Shrines

Possibly the first example of a store implementation utilising page based mechanisms was the Shrines [Ross 1983] store. This system was implemented under the VMS operating system and used the file mapping mechanisms provided by VMS. Shrines used an after-look storage mechanism and operated by directly manipulating the page tables used to describe the process virtual address space.

5.2.2. Page Based Napier88 Stores

Two stores have been constructed to support the Napier88 system at the University of St Andrews using page based techniques. These are the stores described by Brown and Rosenberg [Brown and Rosenberg 1990] and Dave Munro [Munro 1993]. The system described by Brown is implemented using a before-look strategy, that by Munro an after-look.

As described in Chapter 2 these Napier88 systems manage their own area of addressable memory within which objects are both created and mutated. Access to the store is provided by mapping the store into a separate area of virtual memory. Objects are copied to and from this mapped area as required by the Napier88 run-time support system. The method by which these stores are presented in virtual memory and the mechanisms by which the stores are managed is described here.

5.2.3. Browns Before-Look Store

The before-look store described by Brown is presented in addressable virtual memory by file mapping the entire store file as a single entity. The store file is divided into two areas, one that represents the persistent virtual address space, termed the *active space*, and the area in which shadow copies of modified pages are stored, termed the *shadow space*. Because the shadow space is placed immediately after the active space in both the store and virtual memory the store is unable to be extended in size once it is created.

Since a before-look mechanism does not change the mapping of pages in the persistent address space to store locations, a one to one mapping of the entire store file into the process virtual address space can always be used and thus only requires a single call to the `mmap` system call. This has the advantage of not requiring the creation of a large number of individual file maps, each of which requires allocation of kernel data structures. Furthermore, the store does not require a mapping table to describe the correspondence between physical pages and logical addresses. However the store does require a shadow map to record the mapping of modified pages to their original versions.

5.2.3.1. Operation

The operation of the shadow paging mechanism is depicted in Figure 39 below.

1. A running Napier88 program will occasionally copy objects from its local heap back into the store. This operation modifies the data in the active space of the mapped store.
2. When the first attempt is made to modify a page in the active space, the existing data on the page must be copied onto a shadow page and the mapping from modified active page to shadow page recorded within the store. Since both pages are mapped into the processes virtual address space, copying of the data can be acheived by copying from one address range to the other within VM.
3. The copied data is then secured in the store file using the `msync` system call.
4. When a page is shadowed, a new entry is made in the shadow map held within the root block and the root block written to the store

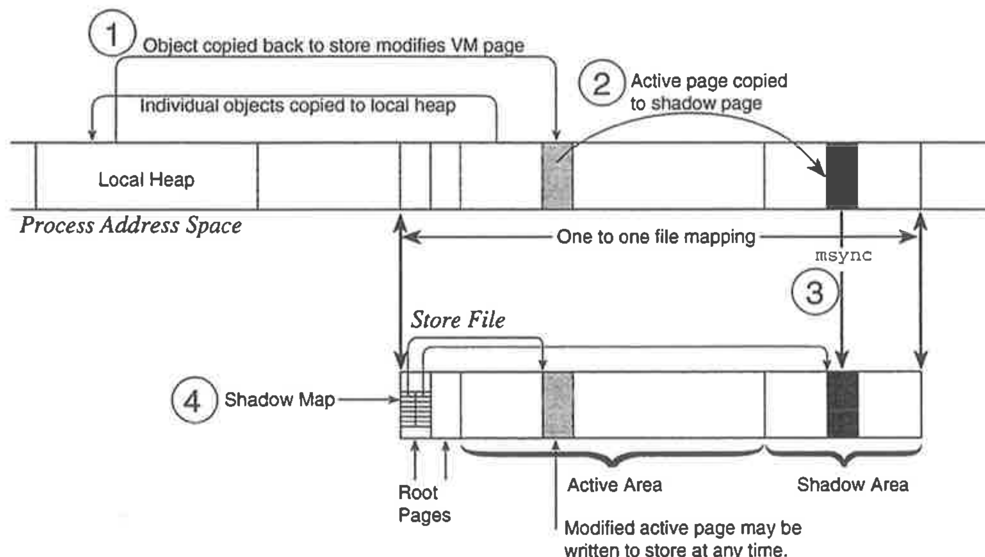


Figure 39. Before look mechanism in Brown's memory mapped store.

Melding the store is achieved by forcing the latest versions of any modified pages to the active space in the store (again using the `msync` system call) and then writing a new root block (using Challis's algorithm) with an empty shadow map to the store. This action implicitly deallocates all those shadow pages previously in use.

When recovering after a failure, the system is presented with a root block containing valid shadow map entries. The system must restore the state of the store at the time of the last meld using the mappings recorded in the shadow map. This is acheived by overwriting those pages in the active area with the data in the shadow pages.

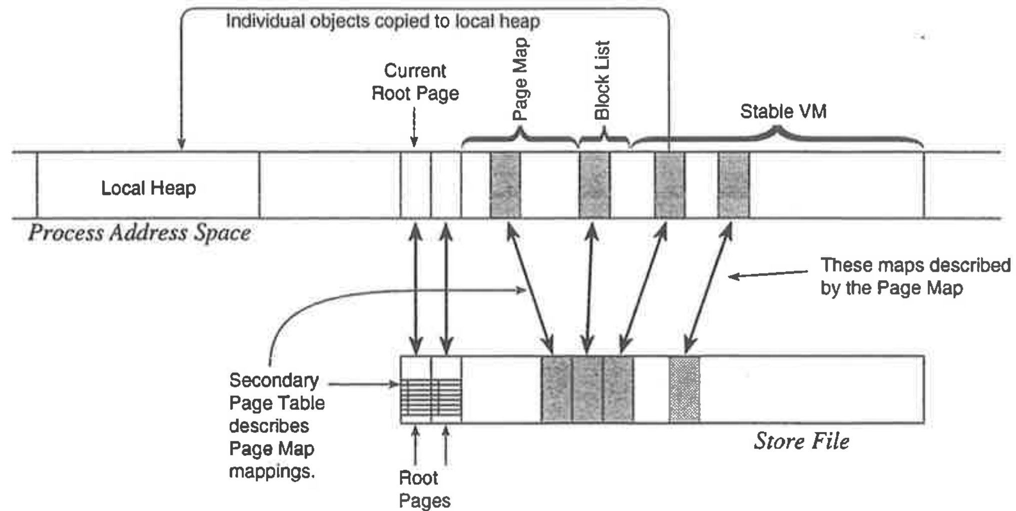
5.2.4. Munro's After-Look Store

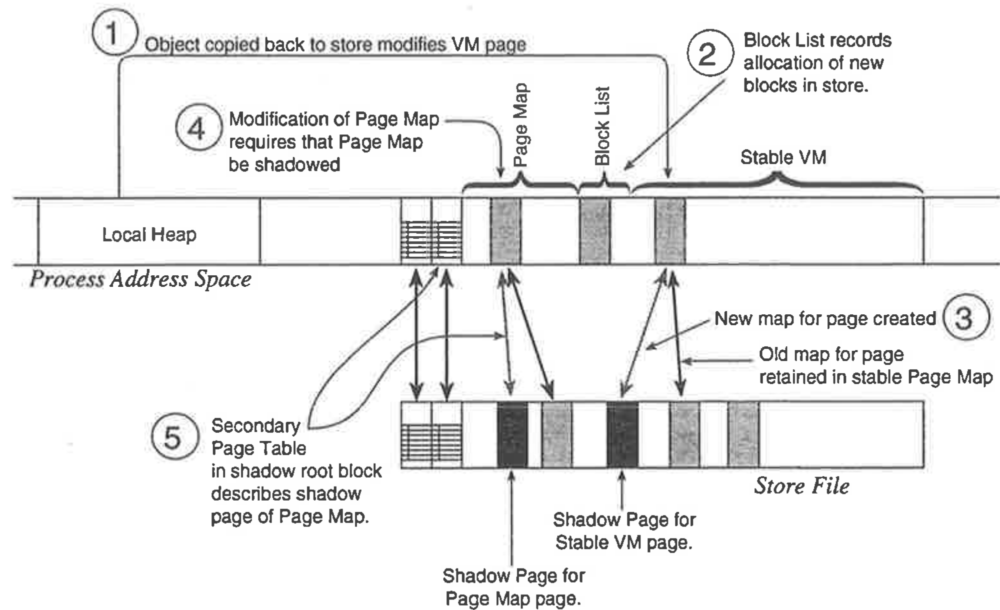
Dave Munro describes an after-look store that supports the same Napier88 implementation as that described above. The store provides the same interface to the Napier88 runtime system, with objects being copied from an area of stable virtual memory presented in the processes virtual address space, into a volatile local heap where objects are mutated and created.

To effect an after-look mechanism a map describing the mapping from virtual addresses to individual pages of file store must be kept. Since the mapping is arbitrary, rather than the fixed one-to-one map of the before-look store, the store file is able to grow after creation.

The page map is presented in virtual memory through mapping in the same manner as data pages used for Napier88 objects. Indeed the pages used to hold the page map could be taken from the same stock of free pages within the store. The page map is a persistent and resilient structure, it must therefore be itself shadowed. The mapping of the page map is maintained in a table (termed the *secondary page table*) kept within the root page. Thus two versions of the page map exist, each described by the secondary page table within one of the two root pages that describe the store. As the root pages exchange roles (though Challis's algorithm) the page map advances in time and through it, the data held in the store. Operation of Munro's store is depicted in Figure 40 below.

An allocation list for pages used in the file is also kept. The information in this list is also available by traversing the page map, but as an implementation optimisation, to improve the speed of allocation of file pages, a separate list is maintained.





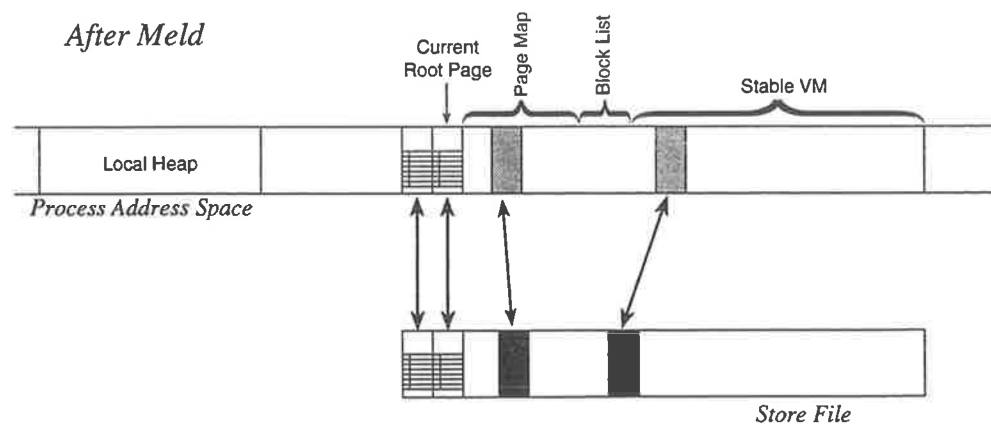


Figure 40. Munro's After-Look store.

Munro also describes a mechanism by which the after-look store can support transactions. This system requires a higher level mechanism to broker transactions. The system used is

the CACS mechanism of Morrison and Stemple [Stemple and Morrison 1992] which provides a generic framework in which concurrency control can be specified.

The store maintains a set of transaction mapping tables which allow separate threads of control to each have their own version of the mapped store. When a transaction commits, the store melds the data in these per-transaction pages with the data within the store. This is acheived safely through clever use of logical operations.

Provided that two separate actions have not modified the same object (something guaranteed by the higher level concurrency control) then the version of a modified page may be melded with the store and the changes safely propagated to the other versions by using a bitwise exclusive-or (XOR) operation. The XOR operator has the following property. If the page as modified by an action A is termed P, the original page O, and the page as modified by a second action B is termed Q. Then Q' may be set to Q XOR (P XOR O). Q' will contain the modifications made by action A but will otherwise remain unchanged. Thus it is possible for any one action to meld and the changes it has made to be safely propagated to other, as yet unmelded actions.

5.2.5. Thatte

Sattish Thatte [Thatte 1986] introduced a variant upon after look shadow store design for use on the Texas Instruments Explorer machines. These machines are similar to the Symbolics 3600 in providing explicit support for the Lisp language. Unlike the Symbolics 3600 the TI Explorer provides a resilient persistent virtual address space in which programs execute. Persistence of objects is provided through reachability from a distinguished root object. A special context object which is automatically filled with the contents of the machine registers when a meld is performed, by arranging to make this object reachable a process can also be made persistent. The design of the persistent store is a variant upon after-look shadow paging and is deserving of study.

In the same manner as other page based stores, the contents of the persistent virtual address space are described by a page table which maps pages in the virtual address space to pages in the store. Since the TI Explorer is limited to an address space of 128MB this table is of modest size.

In Thatte's store each page in the virtual address space is represented on disk by either a single page (termed a *singleton*) or by two pages (termed *siblings*.) Sibling pages provide the functions of a current page and a shadow page, whilst a singleton page is used when the represented data is not changed and does not require shadowing. Each page on disk is time-

stamped when it is written. Time-stamps are derived from a reliable system clock that is guaranteed to advance sufficiently quickly that consecutive pages written to disk will receive different time stamps. Thatte contends that 64 bit time-stamps will always be large enough. Time stamps for individual pages are stored in the page table. Whenever the system melds, the current value of the time-stamp counter is written to the store. It is relative to this time-stamp that the state of individual pages in the store are determined. This is achieved with the following rules.

- Singleton pages with a time-stamp less than the meld time. These pages belong to the store state and contain useful data.
- Singleton pages with a time-stamp greater than the meld time. Such pages were created after the meld and do not form part of the persistent state. They may be overwritten when useful data is created.
- Sibling pages in which both pages are less than the time-stamp. The page with the greater time is the current page and contains useful data, the other is the shadow page and may be overwritten with new versions of the page data.
- Sibling pages with one page earlier then the meld time and the other greater. The page with the lower time is the current page, the other page was written after meld and thus does not form part of the store state. This page is therefore used as the shadow page.

These rules depicted in Figure 41 below.

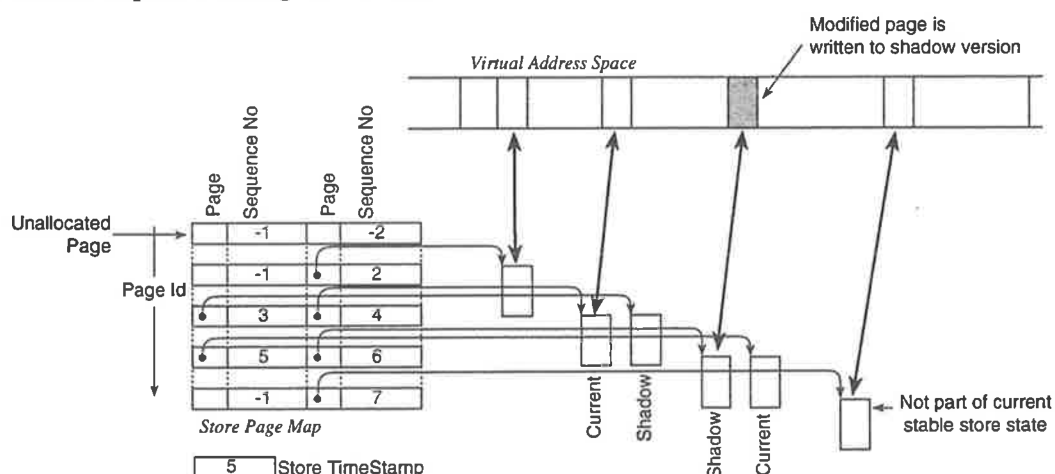


Figure 41. Thatte's shadow store.

The store design integrates the functions of demand paged virtual memory with the persistent store and does not require use of a separate swap space. Pages of data are therefore subject to asynchronous eviction from physical memory into the stable store.

Modified pages are written to their shadow page in the store. Those pages only represented by a singleton page will require the creation of a sibling page to receive the modified data and thus avoid overwriting the current stable version. Pages that remain unmodified for some time may be converted back from sibling to singleton form, thus releasing disk space.

To effect a meld the system writes all modified pages to the shadow pages within the store (updating the time-stamps appropriately). It then writes the current version of the timer to the store. It is the writing of the new timer which defines the atomic movement of the store from one persistent state to another.

The design of the page table is essentially the same as one which incorporates both versions of a conventionally shadowed page table into the one entity. This is doubly unfortunate. The design requires the use of time-stamps to label the entries in the table; these triple the size of the page table. Secondly, because there is only one version of the page table, the design is not resilient against failure during writing of the page table. Corruption of the page table will leave the store unrecoverable unless the page table is shadowed. However such shadowing obviates the need for the dual entry page table design. Although an interesting aside, the design described is only of historical value.

Sattish Thatte also discusses the utility of adding a transaction management package to the system. This is intended to provide a level of fine grained recovery through the use of undo and redo logs that reflect changes to individual objects. It is unclear how his proposal integrates into the persistent environment since it makes the resilience of objects dependant upon explicit use of the package by the programmer. The persistence so created is thus not orthogonal, and conflicts with the persistence supplied by the paging mechanism.

5.2.6. Logging Systems

Logging schemes have also been used in the provision of page based persistent stores. Two such schemes, are the Recoverable Virtual Memory of Satyanarayanan et al [Satyanarayanan, Mashburn et al. 1994] and the Texas system [Singhal, Sheetal et al. 1992]. The Mneme [Moss 1990] system is also of interest in the manner in which it provides a resilient persistent object based store utilising redo logging.

5.2.7. RVM

The design of RVM [Satyanarayanan, Mashburn et al. 1994] grew out of work done in the Camelot project [Eppinger, Mummert et al. 1991]. Camelot is a research system designed to provide general purpose transactional support in an effort to simplify and encourage

construction of reliable distributed systems. However after experience with Camelot, the designers found problems in scaleability, maintainablility and a restrictive programming paradigm. The authors of RVM found the most useful aspect of the Camelot system was its provision of recoverable virtual memory. This experience led them to implement the simplified and more efficient RVM system described here.

The RVM system is provided as a linkable library that may be used by programs running under Unix. Individual applications remain responsible for any distribution and serialisability of concurrent actions. The RVM layer provides an abstraction of memory upon which atomic, recoverable actions may be performed. Data is stored in *segments*, which are ranges of contiguous addresses up to the limit imposed by the system architecture. More than one segment may be used by a program at any time; individual ranges of a segment may be mapped into a processes address space at arbitrary locations. This mapping is acheived through use of the Unix `mmap` system call, although the designers also intend to use the Mach operating system and its external pager mechanism (described in Section 5.2.8 below) in a future implementation.

The user is presented with a group of interface routines which allow transactions to be performed upon designated ranges of memory. Internally the RVM system utilises logging to reflect changes made to recoverable data. A noundo/redo mechanism is utilised for logging. The system is capable of performing log truncation in parallel with execution of the users program; that is, it will concurrently traverse the log applying the changes listed in the log to the database, whilst allowing additions to the head of the log. It also provides for incremental truncation by writing modified pages from addressable memory directly to the original store. Such truncation must be controlled with extra locking of mapped data to ensure that any pages which have been modified by uncommitted transactions are not written to the store before the transactions actually commit.

5.2.8. Texas

In Chapter 3 we described the mechanisms by which the Texas store managed a page based object system supporting the C++ language. Here we discuss the manner in which the persistent store is maintained.

The original Texas store was provided with a redo-log, separate from the store itself, into which copies of all modified pages were written. A log-structured store has since been constructed to provide persistent storage. In a log-structured store the log itself acts as the repository of data. The principal intent of a log-structured store is to reduce disk head

movement as much as possible. This is acheived by writing updates to the store in linear disk block order through regions of contiguous storage. The Texas log-structured store is implemented above a raw Unix disk partition. In the log-structured store, blocks may be written to any location, it is simply the last version written to the store that is the current version. However the store's meta data, describing the location of stored blocks, is also subject to change as logging proceeds. Rather than perform in-place updates of meta-data on the disk (and hence destroying the locality of disk access) the blocks upon which the meta-data is itself resident are treated in the same manner as normal blocks and written sequentially through the log.

The Texas log-structured store uses a tree data structure to represent the locations of blocks within the store. Once the top level node of the tree is written to the store a new stable state is recorded and is recoverable.

The tree nodes are structured as a multi-way search tree (similar to a B-tree) with data blocks as leaves. The tree structure is *right-shallow*, which means that the incomplete right sub-tree of the whole tree is directly referenced by the top level node. Thus the top node directly references recently modified intermediate nodes, reducing the number of nodes that must be written to describe a new store structure. However when the top node fills, a new node must be created to describe further updates. This becomes the new top node and the old topmost node is pushed down one level. Thus the tree grows in depth over time and become increasingly unbalanced. In [Singhal, Sheetal et al. 1992] the authors note that that the tree will require occasional rebalancing, and activity they note is "expensive."

Other difficulties in such a store are concerned with reclamation of the store space and with compaction of the store. The store will eventually fill, this requires that unreferenced blocks be made available for reuse. Texas uses a traversal of the store structure to find the referenced blocks and thus free the remainder.

Compaction is required when the store become fragmented. If the store is allowed to become overly fragmented it will become difficult to keep disk head movement low, and thus will loose the main advantage of the store design.

5.2.9. Mneme

The Mneme store [Moss and Sinofsky 1988; Moss 1990] has been designed to be a scalable system for the exploration of high performance provision of persistent storage. It supports a number of languages including Smalltalk-80 (as described in Chapter 2) and Modula-3 [Kalsow and Muller 1991].

The Mneme store provides access to persistent objects through a *client library*, which is responsible for providing access to objects requested by a user program. The library does this by making requests of *server modules*. It is these server modules which provide resilient persistent storage and are responsible for enforcing security and access control. The client libraries make object data available to the user program in *local buffers*, areas of memory local to the client programs. User programs make requests of the client interface in terms of individual object identifiers, whereas the client to server interface requests data in terms of *physical segments*. The mechanism by which the appropriate physical segment is located from the object identifier is one of the important aspects of the Mneme design. Physical segments in general contain many objects and thus offer a means for clustering of objects during storage and retrieval.

The persistent store is divided into a set of *files*. Files provide a means of dividing up the stored data, and files may be individually compacted. Files may often be named through the host operating system naming mechanisms. File names are not used for object naming, nor are they visible to the programmer, and are thus different to the *databases* of PS-algol.

Objects reside within *pools* which are logical grouping of objects within files. Each pool is associated with a set of routines (a *strategy*) which determines object management policies. The Mneme system allows for dynamic loading of new strategy routines.

Objects within files are physically located within physical segments. A physical segment may contain many *logical segments*. Logical segments are used as a partitioning mechanism in the structured addressing system used by Mneme [Moss 1989]. Physical segments used within the store are the same as those transferred through the client interface and are the basic unit of granularity of Mneme.

Changes to persistent objects occur as part of *transactions* in Mneme. Modified data is returned to the users volatile buffer and thence to the persistent store as part of a commit operation. The current system uses a redo log to record changes made to individual objects within a segment, and it is these logs which provide resilience and crash recovery for each segment and thus the persistent store. Mneme's transaction interface allows the store to determine those objects that have been modified and hence generate appropriate log entries. Hoskins, Brown and Moss [Hosking, Brown et al. 1993] also describe a mechanism derived from the page-cards mechanism described in Chapter 2 for utilising

page protection faults to generate a list of modified objects and similarly generate log information.

5.3. CASPER Bi-Phase

We now examine the implementation of a direct mapped after-look paged based store. This store provides a resilient persistent store which is intended to provide the underlying support for the Napier88 language and its persistent heap architecture described in Chapter 2.

This store architecture has been implemented under Unix, making use of the file mapping capabilities provided by some versions of that operating system, and also under the Mach operating system [Acceta, Baron et al. 1986] taking particular advantage of the external pager mechanism provided by Mach. The Casper store has also formed the basis for a distributed coherent persistent address space system, the design of which is examined in detail in Chapter 6. Further extensions are also described. The first allows the store to create partial melds of subsections of the supported persistent space. The second allows the store to maintain arbitrarily many versions of the persistent state, allowing its use in multi-store distributed systems with external consistency control.

5.3.1. Basics

The Casper store provides support for the same Napier88 language system as that those of Brown and Munro described above. It differs in a number of important respects.

The most obvious is that rather than supplying a separate stable virtual memory abstraction from which objects are copied to a separate volatile local heap, the supported Napier88 system acts directly upon the stable virtual address space itself. This obviates the need for an intermediate layer of support software to mediate object movement. As described in Chapter 2, the notion of a local heap is still used, but local heaps reside *within* the persistent address space. The techniques described in Chapter 2 are used to control object movement and pointer assignment both within local heaps, and within the persistent address space.

The second difference is the manner in which the page table is managed. In Munro's store the page table is itself described by a separate table kept within the root page of the store. The Casper page table is embedded within the persistent address space and is thus self describing.

The Casper store is also extended to provide two extra features. These are:

- The ability to meld only a potion of the supported address space whilst retaining internal consistency. This feature is exploited in the distributed version described in the next chapter.
- The ability to retain more than one self consistent version of the supported address space. This ability is needed when the store is used within a larger framework of cooperating stores where external global management of consistent state in provided. This design is directed at use in the Grasshopper system described in Chapter 7.

These additional features will be described once the basic store mechanisms have been covered.

5.3.2. Page Map.

In Casper, the mapping tables (henceforth known as the *LPMap*, from Logical to Physical Map) are able to describe an arbitrary sized address space (within the limits imposed by the host architecture) and are themselves embedded within the persistent space they describe.

Casper can implement a single persistent store using a number of separate Unix files or raw disks. Each LPMap entry encodes both the file (or disk) and offset within the file (or disk) of the corresponding physical page. Earlier, in Chapter 2, we discussed the use of crossing maps for the description of objects resident upon a page. A natural place to keep crossing map data is in the translation map, the Casper store can be configured to include crossing map information in the LPMap. The LPMap entry format is depicted in Figure 42 below.

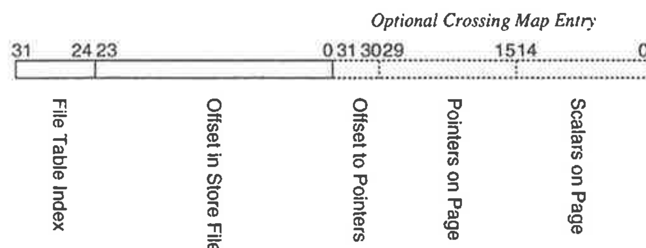


Figure 42. Format of LPMap Entry.

The LPMap is a large data structure that must be maintained in a stable and resilient manner. It is therefore natural to embed the LPMap within the persistent address space that it itself maintains (Figure 43). Once this is done the LPMap becomes self describing and only a single reference to the page which holds the first page of the LPMap (termed the

primary LPMap page) need be held. Only those physical pages actually required to describe address ranges in use are allocated, thus small stores occupy very little disk space whilst very large stores can be seamlessly maintained.

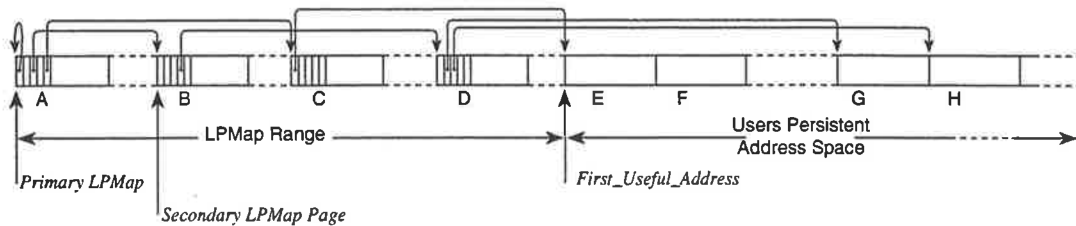


Figure 43. The LPMap is embedded within the persistent address space.

The size of the LPMap is dependant upon a number of choices:

1. Whether crossing map support is required.
2. The target architecture's page size.
3. The target architecture's address range.
4. User requirements for store size.

For 32 bit architectures the choices are not critical. For instance a store requiring crossing map support and targeted toward machines with an 8k page size needs at most a 4M byte LPMap to describe the entire 4G byte address space.

When used with machines supporting larger address spaces more care is needed. A 64 bit architecture using 8k pages and similarly utilising crossing maps requires an LPMap spanning 2^{54} bytes. This significantly exceeds the addressing limits of current implementations of the Digital Alpha AXP architecture, which, whilst supporting 64 bit addressing imposes an overall implementation limit of 2^{43}, or 2^{42} on the user space alone. However to support an address space of 2^{42} bytes an LPMap need only span 2^{23} bytes. It should be emphasised that these allocations only reserve address ranges for use by the LPMap and that no space in the persistent stores is actually required until the address ranges described are actually utilised.

5.3.3. Store Structure

Each Casper store consists of one root file and may also include a number of secondary files. The root file is distinguished by holding some of the special system data structures, otherwise both the root and the secondary store files are identical. Each store file contains a pair of root blocks and a pair of allocation bit-maps sized to describe the individual store file. This duplication of structures within the store files, one acting as a shadow whilst the other contains the current version (with the roles swapping upon meld) is used for all store

data structures apart from the LPMap which is inherently shadowed. The root store contains the following extra data structures.

- A secondary file table. This names each of the secondary files that form part of the store.
- An allocation map for the persistent virtual address space.

The running store system maintains volatile versions of these structures as well as some extra structures discussed in the following description of the store. The internal structure of the running system is depicted in Figure 44 below.

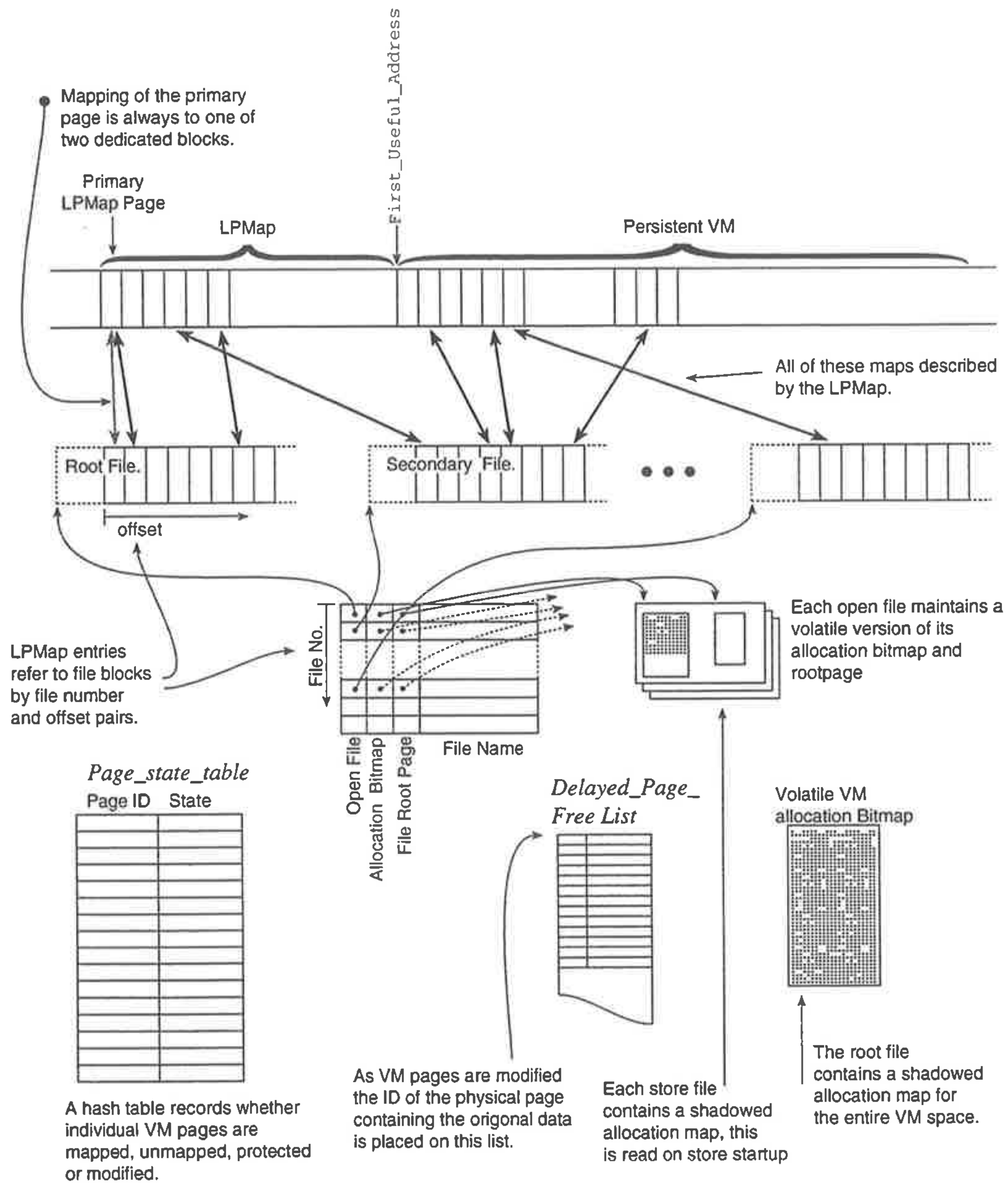


Figure 44. Casper page mapped store architecture.

5.3.4. Store Creation

A store is created by a separate, specialised program which is responsible for creating both new stores and augmenting a store with secondary stores. When a store is created, the two root pages are built, however only one need contain valid information apart from the time stamps, since the other will be initialised appropriately the first time the store melds. The store building program is automatically configured with both the target architecture page size and the disk block size. The disk page size determines the granularity of file space allocation, whilst the architecture page size is required to determine the layout of the initial LPMap entries. An initial store can optionally contain two extra pages. These are:

- A preallocated page. This allows a store to be initialised with a small amount of in initial data, currently a preallocated page is zero filled.
- A zero page, which is available for mapping to any location in the persistent virtual address space which requires zero fill. This is currently only required when the store is used with a Unix based file memory map implementation.

Store files are of fixed size, if a store is being created within a conventional file system the size must be specified, if the store is to be constructed using a raw disk, the store must be the same size as the disk. When created within a file system, the entire file must be built at once. The Unix file-system has particularly poor performance when extending files, and the creation of store files can be lengthy. A large part of this poor performance is due to the mechanism by which files are extended in fragments. By default, fragments are one eighth of a file system block in size and extending a file involves continual recopying of these fragments. It has been found that configuring a file system with a fragmentation factor of one (defeating the fragmentation mechanism) results in an order of magnitude improvement in store creation speed. Stores which involve dynamic extension of allocated space also benefit from this. When a store is first created the layout is as depicted in Figures 45 and 46 below.

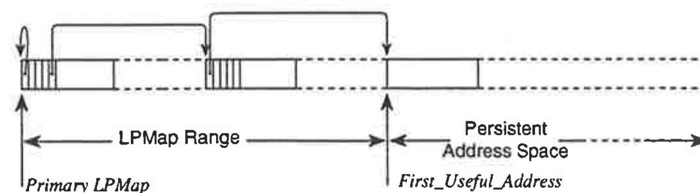


Figure 45. Initial Store Virtual Memory Layout.

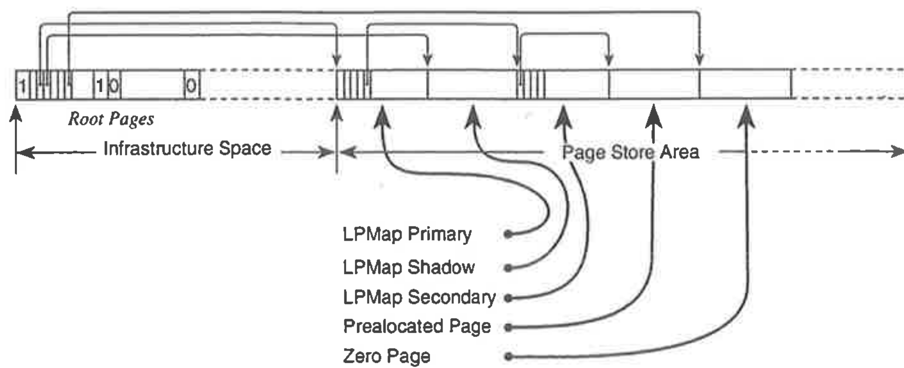


Figure 46. Initial Store Physical Layout.

5.3.5. Secondary Files

Secondary files are also created by the store initialisation program. This program is also responsible for linking the secondary files to the root file. To do this the program must open the root file and write the path name of the secondary file into the secondary file table. This action must be performed in a reliable manner and therefore the update is performed using the same mechanism as store meld.

5.3.6. Store Start-up

When a persistent application begins, it first determines the store to which it must bind. In common with many Unix utilities, the Casper store uses the Unix environment variable mechanism to provide such configuration information. Two environment variables: `persistdir` and `rootfile`, describe the location of the root file of a Casper store.

The first step after locating the store is to open the root file and build the volatile data structures from the contents. First, the two root pages are read into memory. The time-stamps are checked to determine which is the most recent, and to ensure that it is not corrupt. If one root page is corrupt the system falls back to the alternative root page (if possible.) The selected root page refers to the current version of the other data structures in the store, these are read and in-memory, volatile, copies made. Once this has been accomplished the secondary files table is traversed and any secondary files located. Secondary files are opened in a similar manner, they are also described by dual root pages and each contains its own allocation map which is read into volatile memory.

Once all volatile structures have been reconstructed it only remains to build the LPMap. Earlier versions of the store constucted the entire LPMap as an in-memory data structure when the store was opened. This proved to be a very expensive operation and was often the dominant cost of running a program in the persistent environment. To eliminate this process, the LPMap is itself mapped into addressable memory using the same mechanisms

as the rest of the store. However because of the self referential nature of this action care must be taken that the recursive shadow mechanism used for the LPMap has a fixed point.

To ensure this the primary page is immediately shadowed when the store is opened. The page is mapped, a copy made into a buffer, the page re-mapped to its shadow page and finally the page copied back from the buffer. Since this page is now shadowed, its access protection is read and write. Thus the page may be accessed by the page mapping algorithms without the access itself requiring action by the mapping algorithm. All other pages within the LPMap are mapped on demand during normal operation of the store.

The remainder of the virtual address range used for persistent storage must be protected from all access to enable the exception mechanism to detect any access attempts. Under Unix unallocated virtual addresses are already so protected, and no further action is needed.

Since the store implements an after look mechanism no recovery action needs to be performed. The store is always opened in the same manner no matter what its history.

5.3.7. Napier88 Implementation

When used in conjunction with the Napier88 implementation described earlier, the store must be initialised to include the basic structures expected by the Napier run-time system. These structures include the store root object, process table, process proxy table, and predefined constant values. These are installed by the Napier runtime environment during the first use of the store. The Napier environment expects to find the store root object at a distinguished address (the *first_useful_address* supplied by the store layer.) The presence of an object is detected by looking for a valid object header. If no header is found the runtime system automatically builds the default root object and its closure. This is performed as follows.

The Napier88 compiler includes the majority of required objects in each code file it generates. The runtime system copies the transitive closure of the root object in the code file into the persistent address space, following all the pointers in the root object except for the pointer to the code vector. The code vector and its closure is later copied to within the local heap of a newly created process as part of the ordinary program start-up sequence, it does not form part of the default environment. The process table and process proxy table are not part of the compiler generated environment (they are peculiar to the Casper implementations) and are explicitly created by the Casper runtime system. Once this basic object set is resident, the store is able to host Napier88 programs.

5.3.8. Runtime Actions

The store only provides access to areas of memory that have been explicitly allocated. Although the user has use of the entire persistent address space, only those pages that have been allocated from that space are useable. The store will signal an error if any access attempt is made to an unallocated page. This convention is particularly important in the distributed systems described later, but is also a valuable mechanism for aiding in debugging since it immediately traps bad pointer dereferences.

Pages which have been allocated but have no associated page in the store are delivered zero filled. Thus no store space need be allocated to address ranges until the appropriate page in the persistent address space is modified.

5.3.8.1. Page States

The persistent store system maintains a representation of the state of each page in the persistent address space. This is maintained in the *page state table*. The system is only concerned with those pages that are in use by the running user level program, accordingly the page state table is maintained as a hash table indexed by page address. This table maintains the state of all pages including pages used by the LPMap. In the single client system discussed here, there are few states, they are:

Page_Not_Resident: Pages in this state have never been accessed and have no representation in addressable memory. Absence from the hash table represents this state.

Page_Mapped_Read: Pages in this state are available for read only access. No shadow space has been allocated.

Page_Mapped_Write: Pages in this state are available for read and write, they have been modified (or are about to be on resumption of the user level code) and have a shadow page allocated and are mapped to the shadow page.

Page_Read_Modifed: Pages in this state have been modified but are mapped read only to their shadow page. They have had write access denied to enable detection of write access by a language level mechanism such as the page-card pointer quarantine mechanism described in Chapter 2.

5.3.8.2. Read Access

When a read access is attempted on a non-resident page in the persistent address space the following steps take place. This is depicted in Figure 47 below.

1. The user program attempts to read a location on a page.

2. The attempt at access will result in the delivery of an exception by the Unix signal handler mechanism. At this time the user code will be stalled. The exception handler receives the address of the access exception as one of its parameters. From this it derives the page identifier and retrieves the page state from the page state table. As a consistency check it ensures that the page is not currently mapped. Also the exception handler checks the persistent address space allocation table to ensure that the page is allocated. If it is not, it signals an error and exits.

3. The location of the store page is read from the LPMap. If the appropraite LPMap page is itself not mapped this must be done.

4. If the page has a file page listed in the LPMap the file page is mapped to the virtual address space page using the Unix `mmap` system call. The mapping is specified with read only protection.

   If no file page is associated with the page, the page supplied zero filled. The page is mapped read only to the zero page provided by the store. Many persistent address space pages can map this file page at once, since only read only mappings are performed to this page.

5. The state of the page is updated in the page state table to be *Page_Mapped_Read*.

6. The exception handler returns and the operating system kernel allows the user level code to resume execution.

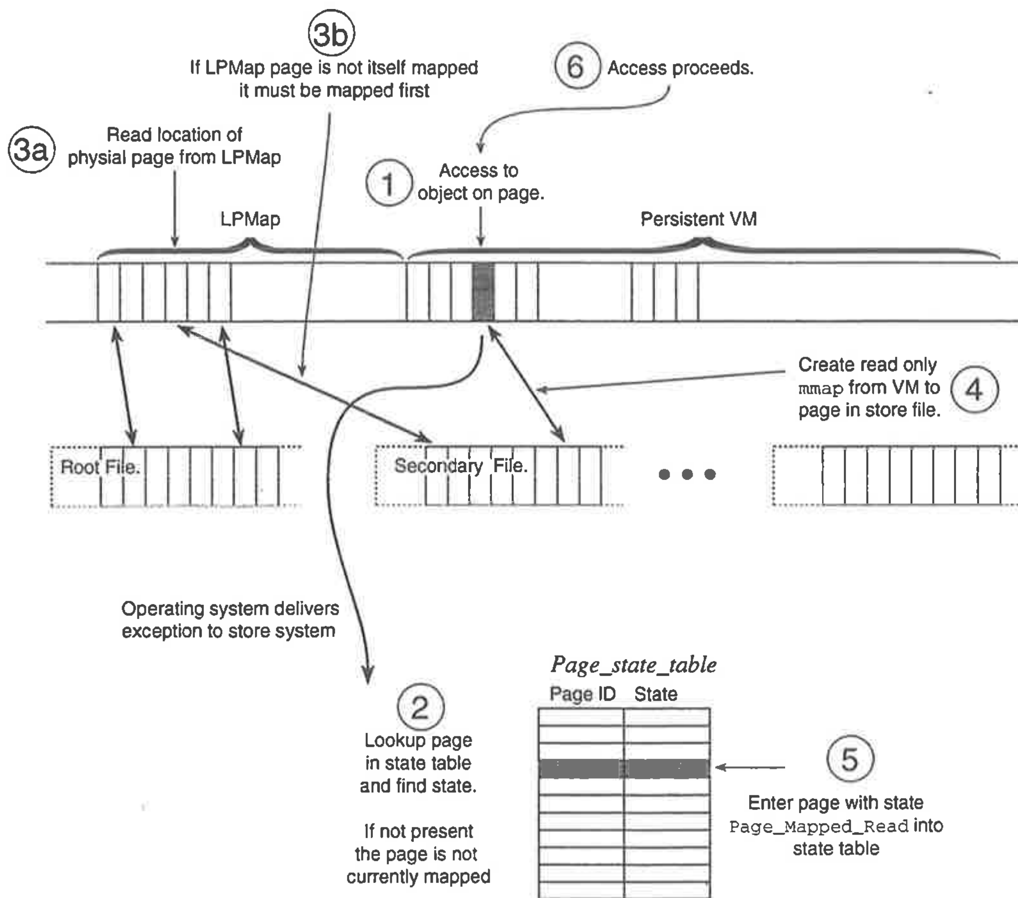


Figure 47. Read access in Casper.

5.3.8.3. Write Access

When a write access is attempted for the first time to a page in the persistent address space the following action is taken. This is illlustrated in figure 48 below.

1. In the same manner as read access described above, the write attempt is traped and an access exception is delivered to a user level exception handler which determines the identity of the page.
2. The page's state is checked from the page state table. If the page is listed as *Page_Not_Resident* it must be first mapped for read. The handler calls the routine *Map_Page_For_Read* which implements the steps 2 though 5 above. The page either now is, or already was, mapped for read. If the page was already mapped for write access a consistency error is signalled and the program exits.
3. The identity of the current file page is determined from the LPMap.
4. This page is entered onto the *delayed_page_free* list. Since this page must not be reused until after the next store meld, it is not deallocated from the file page allocation tables.

5. A new page is allocated from the volatile file page allocation table.
6. The contents of the page are copied to a volatile buffer.
7. The page is re-mapped to the newly allocated shadow page. This mapping allows both read and write access.
8. The contents of the volatile buffer are copied to the page.
9. The state of the page is set to be *Page_Mapped_Write*.
10. The exception handler returns and the operating system kernel allows the user level program to resume execution.

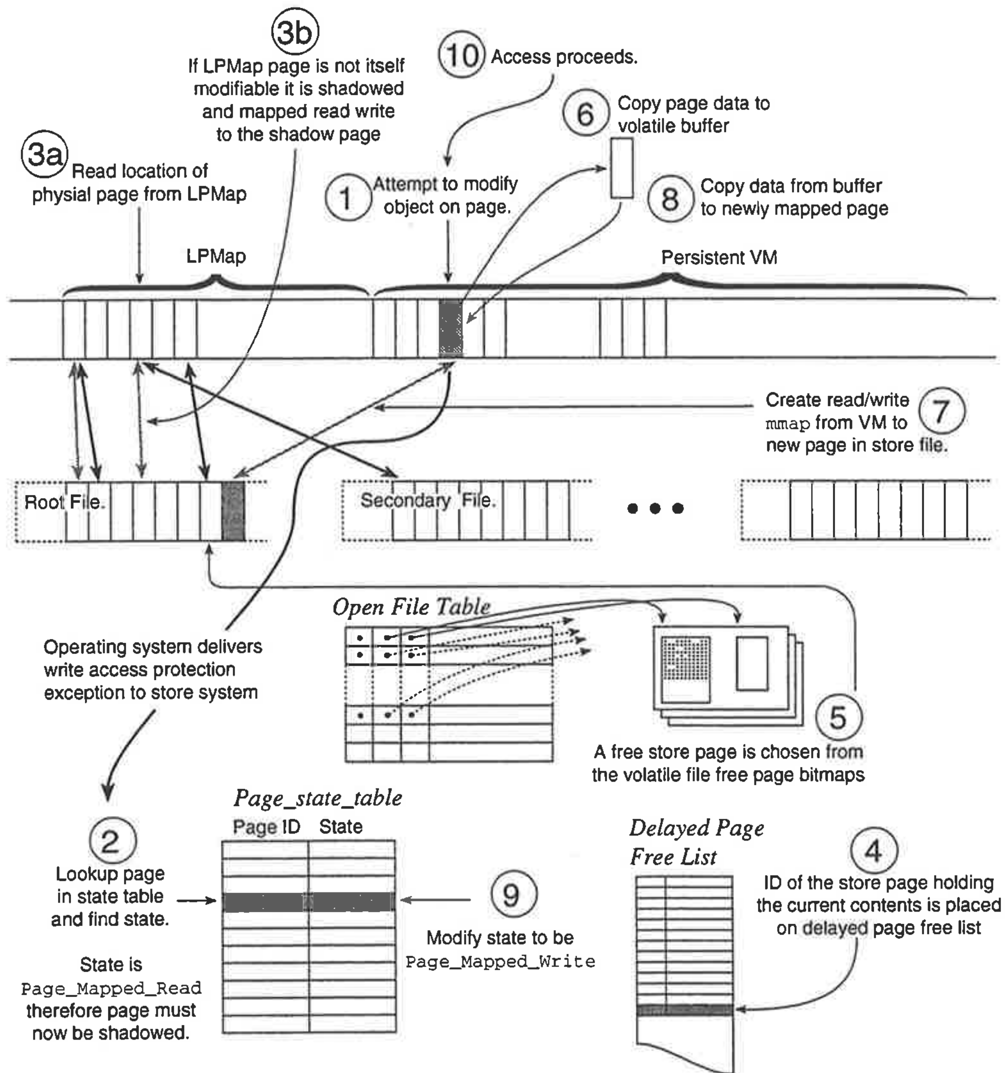


Figure 48. Shadow paging action of the Casper store.

5.3.8.4. LPMap mapping

When the page mapping system requires access to the LPMap, care must be taken to ensure that the LPMap itself is appropriately mapped. Without such efforts the system may deadlock upon access to the LPMap. Two approaches can be taken in providing such availability.

- Recursive invocation of the page map system through the exception mechanism.
- Explicit checking of the state of each page before attempting each access.

Recursive use of the exception mechanism is an appealing choice since with care it can use the same code as described above. However not all operating systems provide for the delivery of multiple exceptions. Unix does, however Mach does not (except though the Unix emulation layer). Furthermore, exception delivery is often implemented using a kernel level stack, one that is held within a pre-allocated and usually small data structure. Although limited in number there can be no guarantee that all the required data structures needed to represent the nested exceptions will always fit. For these reasons recursive exceptions are rejected in favour of explicit checks.

When code that is run from within an exception handler needs to access the LPMap it first calculates the address within the LPMap and checks the state of the appropriate page from the page state table. If the page is not in an appropriate state for the required access a further routine is used to perform the appropriate mapping.

Read access to an LPMap page requires the page to be mapped for read, but to achieve this the intermediate LPMap pages (those that describe the required page) must themselves be mapped for read. When modify access to an LPMap page the page must be shadowed if it is not already. This requires that the intermediate pages are mapped for modify access (so that they may record the shadowing). These intermediate pages will eventually lead to the primary LPMap page. Since this page was explicitly mapped and shadowed when the store was opened, traversal of the LPMap pages will always terminate.

5.3.9. Meld

Creation of a new stable store state is achieved by writing all modified volatile data structures to their shadow locations in the store and then securing a new root page that represents this new data. In practice the different data structures make their way to the store in different ways.

Those data structures which reside in explicitly allocated regions of the store files are written directly to their shadow loctions if they were modified. These structures include the secondary page table and persistent address space allocation tables.

Recall that store pages that contained subsequently modified data are recorded on the *delayed_page_free* list. Although logically free, these pages cannot be reused until the stable state they form part of is discarded. During meld, this list is traversed and these

pages deallocated in the volatile file page allocation tables. These tables now reflect the page allocation of the new stable state. Once updated the allocation tables are written to their shadow versions in each store file.

All those pages modified since the last meld must be written back to their shadow pages. Unix implementations supporting file memory mapping provide the `msync` (memory synchronise) system call which ensures that modified mapped data is forced back to the mapped file. Some pages may already have been written back as part of normal operation of the operating systems paging mechanism. Such pages will not require writing to the store again, thus reducing the load during meld. The store mechanism relies upon the operating system maintaining its own records of pages that have been written, and assumes that only those pages that still require writing to the store actually will be.

Memory synchronisation requests are made for extents of virtual memory. However the allocation of file pages to virtual memory pages is unlikely to be straightforward. Simply traversing virtual addresses in order and writing their contents to the store is likely to result in poor locality of access to disk blocks and subsequent reduction of disk performance. To alleviate this, a list of all modified pages is sorted by physical file page and memory synchronisation calls are made in this order. Although the list is sorted by file page, contiguous runs of virtual addresses are still relatively common since large blocks of memory are often allocated together. This results in contiguous blocks of virtual memory associated with contiguous blocks of disk. Such contiguous runs of virtual addresses are coalesced into single `msync` requests, which lowers the number of system calls made.

Once all modified persistent address space pages are written back to the store, it only remains to write the root pages of each store file to their shadow root page locations using Challis's algorithm. The root page in the store root files is written last. The Unix `fsync` call is used to ensure that all pending disk writes are completed before continuing. At this moment two self consistent versions of the state of the persistent address space exist in the store. The system now allows the user level program to continue execution. As the user program proceeds those pages released from the previous stable state will be reallocated and overwritten, destroying the penultimate stable state.

5.3.10. Recovery

Since the store implements an after-look, noundo-noredo mechanism no separate recovery mechanism is needed. Whether the store is used after a normal shutdown of the system or

after a system failure exactly the same actions are taken. Should the system start up after a system failure it will find a self consistent state in the store which it can immediately use.

5.3.11. Implementation Specifics

So far the description of the store mechanisms has been provided in terms of an implementation using a generic version of Unix, and has avoided specific implementation details. Here we address some of the specific details and problems of implementation under Sun Microsystem's version of BSD Unix, SunOS, and a second implementation under the Mach 2.5 operating system.

5.3.11.1. Placement in memory

The LPMap is logically placed at the beginning of the persistent virtual address space. However this conflicts with the standard memory layout of a Unix process, which places the user code and program state at the beginning of memory. To avoid conflict the LPMap must be placed elsewhere, avoiding both the user program space and the persistent address space.

Likewise the start of the useable persistent address space must be located to avoid both the moved LPMap and the user program space. In current implementations the *first useable address* is set at 8MB and the LPMap located at 4MB. The size of the persistent address space is limited so that it does not overlap the area of virtual memory utilised by the C language run-time stack (which grows down from high addresses.)

5.3.11.2. Page copy sequence

In the description of the page modification sequence above, a copy of each modified page is made before re-mapping the page to a shadow location within the store file. This copying is unfortunate, but is dictated by the lack of ideal support for shadow paging in Unix. The need to copy the contents comes about because when the page in virtual memory is mapped to the shadow file location, the operating system discards the current contents of the page and replaces them with the contents of the file page. This is unfortunate for a number of reasons.

1. The operating system has discarded the in memory copy of the page contents, forcing the store mechanism to make a copy to avoid losing the data.
2. The operating system will use a disk read operation to retrieve the contents of the shadow page at the first access to the page after the new mapping is established.

This is wasted effort since the contents of the shadow page are of no interest and the first action of the store system will be to obliterate this data.

3. To provide an appropriate system call is very simple and requires less effort on the part of the operating system. Such a call was proposed during initial specification of Sun Microsystem's System V derived version of Unix, Solaris 2.0 but was discarded.

   This was the `mremap` system call, which would remap an already mapped page to a different location within the file, whilst retaining the contents of the virtual address range. Clearly this is the desired mechanism.

5.3.11.3. Memory Allocation and Swap Space.

When memory is allocated from within a processe's virtual address space, the Unix kernel automatically allocates space for demand page swapping on disk. Therefore it is not possible to pre-allocate large regions of the processe's address space for use by the persistent virtual address space without causing swap space allocation. This allocation serves no purpose since the store mechanisms will actually perform the role of swap space. Furthermore, attempts to allocate space for very large persistent stores will fail because the system swap disk is too small to hold the required range.

Faced with this restriction great care is need to ensure that addresses within the persistent address space are not accidentally used by parts of the runtime system or operating system. SunOS version 5 allows the user to allocate space with a *mmap* system call with the option of specifying that swap space should not be allocated, thus repairing this problem.

5.3.11.4. Implementation under Mach

A second implementation of the store has been completed under the Mach operating system. This implementation is directed at support of the Casper distributed persistent environment discussed in Chapter 6, but some important points should be made here.

Rather than provide the file memory map mechanisms discussed above, Mach provides a separate abstraction which allows the user considerable freedom in building virtual memory based systems. This is the *external pager*. A user is able to provide their own server code which will be activated in response to page faults occurring in a served region of a virtual address space. The external pager interface also provides a mechanism through which an external pager can protect regions of served memory from access and receive

notification of access protection violations to pages separate to the access protection within each virtual address space. In more detail the Mach external pager abstraction operates as follows.

5.3.11.4.1. External Pager

Mach provides an abstraction termed a *memory object.* A memory object is an abstraction over storage. Each memory object represents a contiguous range of memory which can be accessed by mapping into a process's virtual address space. To a user process access is similar to mapping a file. The operating system kernel is responsible for maintenance of each processes virtual address space and the attendant virtual memory control data structures. The management and provision of the data visible in the memory object is the responsibility of the external pager.

A memory object is identified by a communication port, and it is to this port that the kernel will direct requests for the provision and management of data. The Mach system uses a predefined interface specification by which the external pager and kernel communicate. Communication is effected by the exchange of *messages*. The Mach system supplies an interface library [Draves, Jones et al. 1988] which translates these messages into an RPC based interface suitable for direct use by C programs. This interface provides the following kernel to external pager messages:

- Requests to supply the data resident upon a given page,
- Notification of violation of access restrictions on a given page,
- Return of the data resident on a page,
- Notification of completion of manager request, and
- Sundry housekeeping notifications.

The manager may make the following requests upon the kernel, through a similar RPC interface:

- Protection of a region of the memory object against access,
- Request for the return of data resident in modified pages, and
- Request to place data into memory visible to client processes.

The external pager mechanism offers some important differences from the file map implementation described above, these are:

- Splitting the memory access exception model. As described, Mach provides two separate models of access exceptions: those relative to a process virtual address space and those relative to a memory object. This splitting allows implementations to avoid the intermingling of programming language level use of exception (such as the page-card based garbage collection and pointer quarantine systems described in Chapter 2) from the access detection used to manage the shadow store. Thus the language mechanisms can be implemented independently of the store mechanisms.
- Low level control of page state. Memory managed by an external pager is not paged by the systems default paging system. Thus it is possible to integrate store management with page swapping and avoid the unnecessary extra work that occurs in conventional systems.

5.3.11.4.2. Problems with Mach

The structure of the message passing mechanism used by Mach makes transmission of pages of data to the kernel inefficient. When a page of data is prepared by the external pager (usually in response to a request from the kernel) it resides within the virtual address space of the external pager. This data may not necessarily be page aligned and since it forms part of the external pager's virtual address space would normally be expected to remain visible to the external pager after it has been provided to the kernel. The kernel is thus required to make a copy of the data onto a new page which it then links into the client processes address space. Clearly if the manager guarantees to page align the requested data a system could be designed in which no copy of data is needed. Such considerations have guided the design of newer user level memory management systems in other operating systems [Lindström, Dearle et al. 1994].

When faced with pressure upon allocation of physical memory the Mach kernel will elect to flush the data resident on some physical pages. Modified data on pages associated with memory objects will be asynchronously returned to the external pager. Unmodified data is simply discarded. Lack of consultation with the external pager before carrying out these actions can cause some problems, in particular they complicate management of a distributed shared persistent space. Such problems are addressed more fully in the next chapter.

5.3.12. Partial State Meld

A version of the Casper store has been constructed that allows separate sections of the persistent virtual address space to be melded with the store independently. This store is designed to support the distributed Casper system described in the next chapter. When melding subsections of the address space it is important that no interdependencies exist between the melded subsections and the other subsections. Controlling interdependencies introduced by the user programs is described in the next chapter. Managing those that occur through the store structure is described now.

The distributed Casper system allows independent clients to have access to the persistent address space, each client may mutate data in the space and may allocate address ranges from the space. The store system is modified to ensure that no interdependencies are created during the meld phase. This is acheived as follows.

The file page delay list from the single user store is split into per-client lists of modified pages. When a client melds only those pages within the store freed by that particular client are made available for reuse.

The allocation bitmap for pages in the persistent address space is handled differently. The bitmap is duplicated, one version is termed the *eager volatile* the other the *stable volatile* map. The stable volatile map is a copy, in volatile memory, of the allocation map that is resident within the stable store state. These structures are illustrated in Figure 49 below.

The eager volatile map is used to allocate address ranges to running client programs. It is pessimistic, in that it lists the allocation of pages by all clients but any release of address ranges is deferred until the client melds. This ensures that if a client frees space and later fails without making the release of space stable, another client cannot inadvertently allocate that space and so lead to conflict. The store maintains a per-client log of both allocation and release actions. When a client melds this log is traversed. All requests are applied to the stable volatile bitmap (both allocation and release) thus ensuring that only the changes to the allocation of the persistent address space made by the melding client are reflected in the stable store. The changes are also applied to the eager volatile bitmap, thus releasing address ranges for use by all clients. If all the clients were to meld together both the stable volatile and eager volatile allocation maps would be coincident.

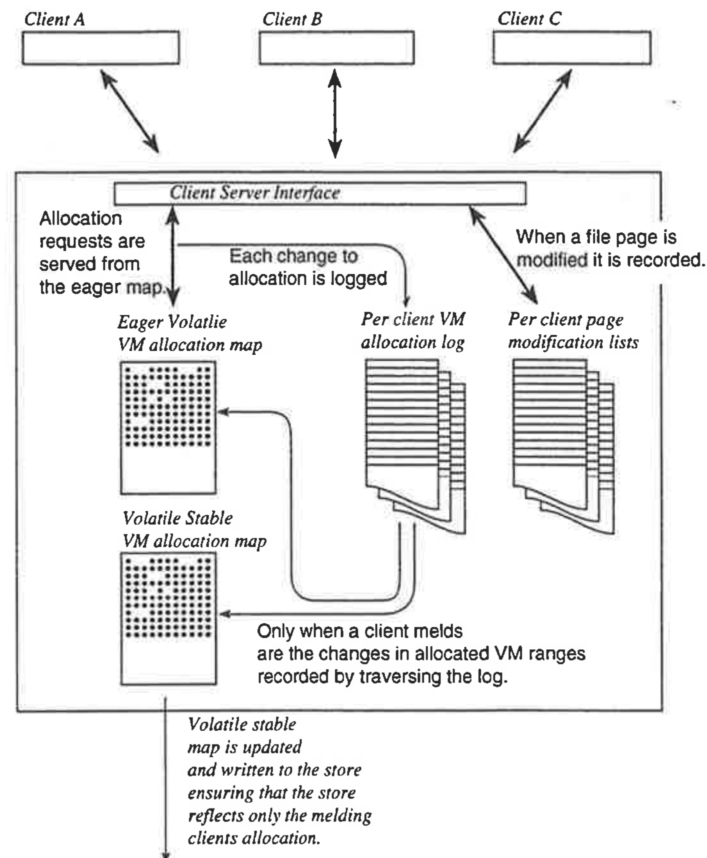


Figure 49. Handling VM allocations in Casper.

The LPMap is also modified for similar reasons. Since the LPMap is placed within ordinary pages, any single page may contain entries reflecting the action of any number of clients. When one of these clients melds the system must ensure that the operation of the other clients is not placed into the stable store in such a way as to result in an inconsistent representation of the stores operation. The LPMap is modified to contain two physical page references for each entry. In a manner similar to Thatte's store one of these will contain a reference to the current page, the other to the shadow page.

To determine which of the two entries is the current and which the shadow entry a separate bitmap is maintained. It contains one bit per LPMap entry and its value toggles the use of the entries in each individual LPMap location. When a page in the persistent virtual address space is shadowed the location of current version of the page is not overwritten, rather the alternate location is overwritten. Thus the LPMap will still retain the mapping information appropriate to the previous time the particular client that is using the page descibed by that entry last melded. The role of the entries in the LPMap for the running programs is reflected by a volatile selection bitmap. When a client melds it is only those entries in the selection bitmap which correspond to pages over which that client has jurisdiction that have their selection bits inverted, thus reflecting the new roles the pages

have. The inverted entries are identified by traversing the modified page list associated with the melding client. Thus the LPMap can be written to the store safely and correctly reflect the changed roles of only those pages pertinent to the melding client.

5.3.13. Multi-Phase

When incorporated into a system of multiple distributed stores with external consistency management it is valuable to be able to maintain more than one self consistent system state within the store. The Casper store can be configured to provide this functionality with very little change to its operation. The operating of this store is depicted in Figure 50 below.

The key to providing multiple states within the Casper store is in the integration of the LPMap into the persistent data space. Because a self consistent store view is always reached through the primary page of the LPMap, maintaining multiple views requires little more than the ability to maintain a list of LPMap primary pages, rather than the two maintained by the store described above. The operation of the shadow paging mechanism results in a system where only the minimum of data is needed to represent each system state through time in a seamless and natural manner. This is expounded upon below.

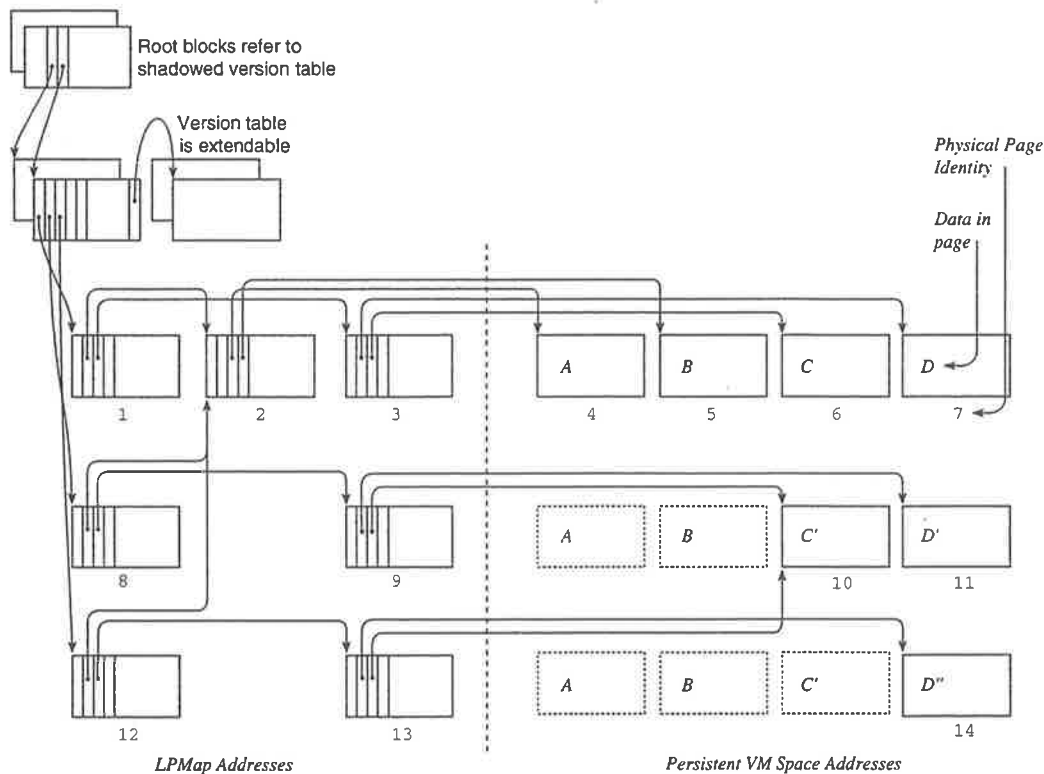


Figure 50. Multi-phase Casper store.

5.3.13.1. Store Modifications

Implementation of the multi-phase store is a natural extension of the existing bi-phase design. In the multi-phase store the allocation bit-maps describing the allocation of file pages and pages within the persistent address space are no longer kept in the store as separate data. These structures do not represent any information that cannot already be derived from the LPMap and retaining them would require that a new version be built for each separate store state. Deletion of these structures increases the cost of store start-up but improves the speed of store meld.

However, once the persistent address space allocation map is removed it is necessary to find another mechanism to indicate that a page is allocated but currently has no store associated with it. Recall that this condition is useful to indicate that a page is to be supplied zero filled without requiring the allocation of actual store space. Zero fill is now indicated by placing a sentinel value into the LPMap entry associated with such a page.

5.3.13.2. Start-up

Start up of the multiphase store is effected in largely the same manner as the bi-phase store. The differences centres on three main issues:

- identifying the appropriate version,
- lack of in-store allocation tables, and
- recovery of unneeded store.

The store is still described by a pair of root pages, and atomically moves from one version to the next through swapping between these pages using Challis's algorithm. Thus the latest version of the store meta-data is found in the same manner as the bi-phase store. However there is no longer a notion of a single current store state, rather a multiplicity of states. The store identifies each of these states using a discrete index number which identifies a primary LPMap page. The store maintains a pair of tables (current and shadow) which contain the index number, LPMap primary page pairs. Upon presentation of an index the system reconstructs the associated stable state and presents it in addressable memory.

Since the store no longer maintains copies of the virtual address space and file allocation tables, volatile versions of these must be built by traversing the LPMap. In principle this could be achieved by simply scanning the LPMap from start to finish. A linear scan may be costly, especially for stores that support addresses of greater than 32

bits. Rather than performing a linear scan, a more efficient method is to recursively traverse the LPMap, beginning with the primary page, and only read those LPMap pages that actually represent allocated pages in the persistent address space.

After a particular state has been loaded those states generated at a later time than the selected one are of no further use. These states represent computations that must now be considered to have never happened. Similarly those states generated at an earlier time are also of no use, they represent earlier states that are now inconsistent with the current system state. All pages that are part of these unneeded states and not part of the restored state will not be referenced from the restored LPMap, hence these pages will not appear in the volatile allocation structures and will be subject to reuse during subsequent computation. Once the store is modified after start-up, the primary LPMap pages that represent these states will potentially refer to pages overwritten by the proceeding computation. The primary LPMap pages are therefore deleted from the version index table and the table atomically melded with the store. To enable debugging of the store these extra versions are not deleted if the store contents are only read (such as might occur when performing an object level sanity check.)

5.3.13.3. Normal Running

During normal operation the store operates in precisely the same manner as the bi-phase implementation. The volatile data structures, LPMap and operating principles are unchanged.

5.3.13.4. Meld Protocol

When the store melds an index identifying the new version is generated by some higher level control regime. This index is written to the shadow version of the index table along with the page identifier of the primary LPMap page for the current version.

Data from the persistent address space makes its way to the store in the same manner as the previously described store systems, through use of `msync` calls if implemented under Unix, by the action of the external pager if implemented under Mach. Since there are no longer any in-store allocation maps, no melding of the volatile allocation structures is required.

Once the meld has completed a new shadow page must be allocated from within the store file and immediately mapped to the in-memory LPMap primary page. This step is

analogous to the eager creation of a shadow for the LPMap primary page at system start-up and is performed for the same reasons.

5.3.13.5. Snapshot Deletion

Deletion of store states is initiated by some higher level control system which manages the global state of the system and the inter-relation of the component stores. Such mechanisms are discussed in chapter 7. Once some such control system has determined that a particular store state is no longer useful, it requests the store manager to delete it. As described earlier, store states are identified by an index number. The goal of the deletion mechanisms is to deallocate all those pages associated with the indexed store state, whilst leaving those pages common to all other store states. A request to delete a store state can proceed in two ways:

- If the store is not in use it is enough to delete the index, LPMap pair in the index table. Modification to the root page must of course be performed in a resilient manner and Challis's algorithm is again pressed into service.
- If the store is running, it is important that those store pages currently allocated to the target store state be immediately made available for reuse. This will be the only way in which a running store can reclaim store space.

  State deletion in a running store proceeds as follows.

  1. The store uses the index to identify the LPMap primary page that is the base of the state to be deleted.
  2. Two further store states are identified: the one that directly precedes the indexed state, and the one directly following the indexed state. The LPMap primary pages for these states are also found.
  3. The closure of pages described by the three LPMap root pages are traversed. Those pages that are common to the target state and to either the preceding or following states cannot be reclaimed. Those pages that are not common may be reclaimed. Pages identified for reuse are added to a list of candidate pages. Notice that since the LPMap is self describing, LPMap pages themselves also form part of the set of potential reclaimed file pages, thus no separate action is required to reuse LPMap pages. Since the LPMap primary page is never in common with any other state it will always be reclaimed.

4. The target state is deleted from the version table. The table must be committed to the store ensuring that the deletion of the version is stable before reuse of the released pages can proceed. Once this is done (through Challis's algorithm as usual) the pages listed for reuse are deallocated in the volatile allocation bitmap and become available to the running program.

5.4. Comparisons and Conclusions

In the store architectures discussed there is a clear progression in the nature of the data structures used to map physical pages to addressable memory. Brown's store does not use such a structure (although the shadow map structure is required, and thus it is not true to say that no mapping data is kept at all.) Munro's store uses a separate mapping structure which is shadowed through the action of a separate description mechanism. Thatte's store uses an unshadowed map structure utilising time stamps to represent the use to which a particular physical page is being put.

The Texas log structured store provides a self describing map structure, however one in which the map is structured as a search tree rather than a conventional page table. The bi-phase Casper store provides a self describing shadowed page map, and the multi-phase Casper store extends this to include multiple self describing maps which can share those parts of the map and store which are common to individual versions.

5.4.1. Log structured versus shadowing

A comparison between a log structured store (such as Texas) and the shadow paging schemes is interesting. The Texas store achieves locality of reference for physical blocks in the store by writing to blocks chosen in sequential order on the disk. The map data structures are also so written, preserving this locality. Shadow paged stores as described thus far, are not able to make use of such optimisations.

However, in Unix based implemetations of Texas, because each page of data must be unswizzled before it is written to disk the system is unable to map the data referenced by the user program directly to the store. The system is therefore unable to integrate the store mechanism with paging by the virtual memory system to swap space. Thus a hidden cost of the system is movement of data pages to and from swap space in response to pressure on physical memory. This movement suffers from exactly the random disk head movement that the log structured store is designed to avoid. This problem must severely limit the

performance advantages found in practice unless the running system rarely exceeds its physical page set size. Implemetations of Texas under Mach avoid this problem

The same effect of disk locality provided by the log-structured store *can* be achieved with after-look shadow stores. The key lies in only allocating the disk block to which a page is mapped at the moment that the page is either evicted from addressable memory or melded with the store. Store designs which use the Unix `mmap` mechanism are unable to do this because the operating system performs evictions asynchronously, forcing allocation of a store block as soon as the page data is modified and the map performed. Implementations which use an integrated store manager, such as the Mach external pager, are able to delay the allocation of the page until it is actually written, thus allowing the allocator to pick the page best able to preserve high locality on the disk. Since the Casper design integrates the page map into the same paging mechanism as data pages, map pages can also take advantage of the locality gains. The rest of the store mechanisms operate in exactly the same manner as before. Unlike the Texas store there is no need to separately traverse the store to find free space, nor does the store space used grow monotonically until such a traversal is performed.

Care must be taken if the operating system chooses to evict an LPMap page from memory. To write this page to disk would require that all the pages it references be allocated immediately. Clearly it is better to avoid eviction of such pages whenever possible. A Mach external pager can simply elect to write a different page to disk and reinsert the LPMap page into physical memory. However virtual memory management designs which provide the pager code with some control over the choice of evicted pages are clearly superior.

Chapter 6. Distributed Casper.

6.1. Introduction.

This chapter examines the design and implementation of the distributed Casper system. This is a persistent system designed around the paradigm of page based distributed shared memory. It draws together the user level view of a persistent virtual address space as discussed in chapter 2, above a page based stable store as described in chapter 5. It augments these by allowing multiple programs to execute concurrently within the persistent space and allows these to execute on physically separate nodes on a network.

Casper supports the execution of Napier88 by clients which connect to a central persistent store. The architecture implements a resilient page-based coherent persistent address space. In this address space, any read operation will always read the result of the last write to the object concerned, independent of the client on which the read or write operation takes place.

6.2. Execution Environment.

The system described in this chapter is implemented using the Mach 2.5 operating system. Mach [Acceta, Baron et al. 1986] provides some important advantages over conventional operating systems. The major features are:

- the external pager,
- inter-process communication
- multiple threads

Under Mach, the user is permitted to provide a process called an *external pager* which services page faults. If an external pager is associated with a user process, the Mach kernel will forward page fault exceptions to that external pager. One can arrange, for example, that the required data will be supplied (in the case of a read fault) or written to some stable medium (for pages removed from the client's physical memory). This external pager mechanism implements most of the functionality needed to support the coherent persistent address space described in this chapter.

The single inter-process communication (IPC) facility available in Mach permits a transparent interface to be built, independent of the physical location of the communicating parties.

Mach supports more than one thread of execution in a single virtual address space, an aspect which is exploited by the architecture described. This is especially useful in building

asynchronous communication protocols, such as the cache coherency protocol described here.

6.2.1. Stability and Coherency

Database management systems require a sequence of update operations by a user process to be contained within an atomic transaction. That is, either all modifications are completed, or none are made. Traditionally, such atomicity is achieved by locking portions of the database, so that while a transaction is in progress, no other user process may view modified data [Eswaran, Gray et al. 1976].

Napier88 provides no language-level synchronisation primitives; however, suitable language features have been proposed for the Napier family of languages [Morrison, Brown et al. 1989]. This proposal does not force all data accesses to be serialised; instead, anarchic access to the store is permitted. However, the persistent store itself must be kept consistent, which presents two problems:

- it must be possible to *roll back* the effects of a series of updates during which a failure occurs (either of a client or the server), and
- no process must view or act upon out-of-date data.

In the architecture described in this chapter, store stability deals with the first aspect, and the cache coherency protocol with the latter. These are discussed in Sections 6.4.1 and 6.6.

6.3. Overview of the Architecture

The architecture of the distributed Casper system is depicted in Figure 51. A number of clients execute against a shared stable store using a coherency protocol that guarantees data integrity; client code executes in an environment that is robust and guarantees correct execution regardless of the failure of parts of the system.

A *stable store* is defined to be a set of objects which move from one consistent state to another atomically. In Casper, the stable store is provided via the *Stable Store Server*. The Stable Store Server consists of four components: the *Server Request Handler*, the *Stable Store Manager*, the *Stable Store Garbage Collector* and the *stable medium*. The objects resident on the stable medium are managed by the Stable Store Garbage Collector, whilst the physical pages are managed by the Stable Store Manager. Finally, the interface to the outside world is provided by the Server Request Handler.

In addition to the usual passive data found in traditional database and file systems, the stable store contains active data including the state of all processes executing within it. This provides the potential for restarting processes found in the persistent store should some element of the system fail. This protocol includes the maintenance of structures needed to correctly roll back the execution state of interdependent clients should failure occur in any part of the system. Those parts of the system that can continue without jeopardising the integrity of the stable store are unaffected. The stable store server is described in more detail in Section 6.4.

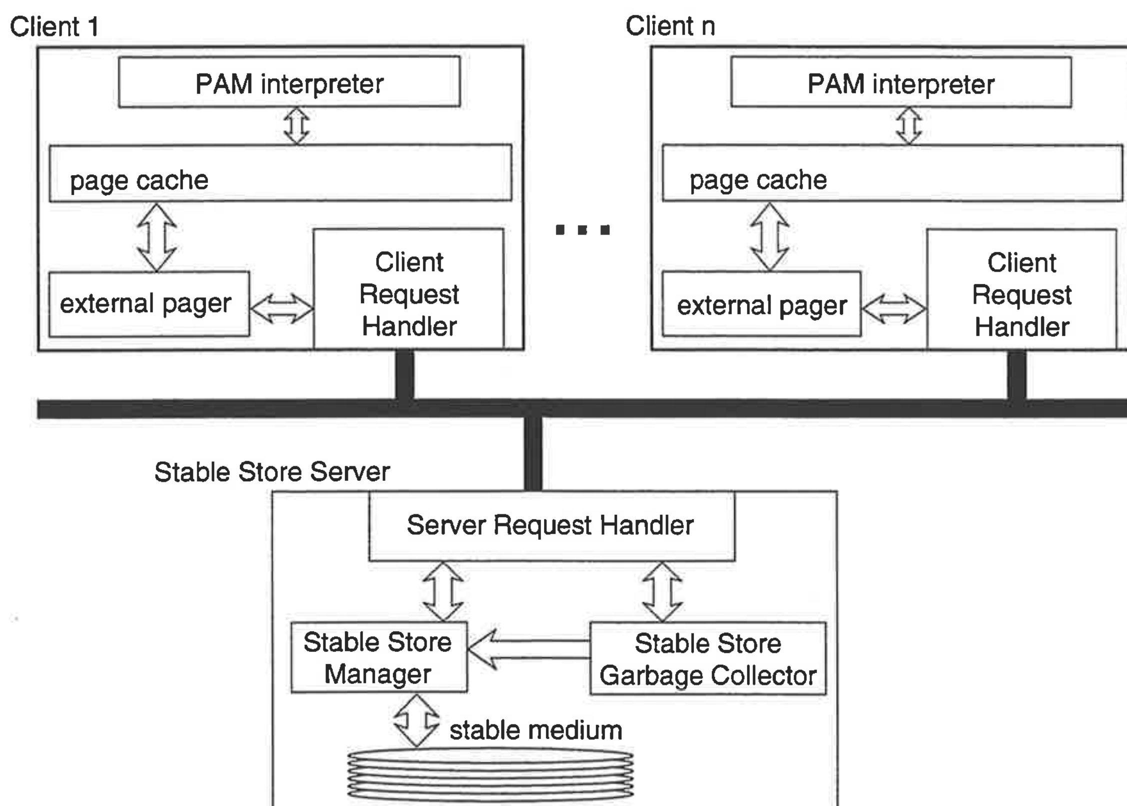


Figure 51. The Casper distributed persistent architecture.

Each client has an interface to the Stable Store Server, which gives access to the stable persistent store; this interface is called the *Client Request Handler*. In addition, each client contains an interpreter for PAM code (such as compiled Napier88 programs) and a *page cache* which holds copies of those stable store pages required by the interpreter. Within a client, coherency is maintained by the Client Request Handler and the external pager.

In Casper, object pointers refer directly to the addresses of objects within the persistent address space, references to nonresident objects are handled as page faults. The external pager services all page faults for nonresident pages; these are fetched from the persistent object store under the control of the coherency manager.

Garbage collecting a large object space such as the one supported by our architecture is potentially an expensive operation; the implementation therefore uses tactics to make garbage collecting the entire persistent store infrequent. One tactic is to maintain an area of locally-created objects to which there are no external references and which can therefore be garbage collected independently; this technique was described in chapter 2.

The coherency protocol is distributed and has been designed as two finite state automata (FSA), one specifying the state of a page within each client, the other the state of a page in the Stable Store Server. In practice, the coherency protocol is implemented by tables – an Export Table in the Stable Store Server and an Import Table in each of the clients. The cache coherency protocol is discussed in Section 6.6, where the operation of these tables is explained more fully.

6.4. Stable store server

6.4.1. Store Stability

As described in the previous section, all access to the persistent store is controlled by the Stable Store Server. It has two main functions: managing the supply of pages upon demand to clients, ensuring that coherent versions of the pages are supplied; maintaining the integrity of the Stable Store. It also allocates ranges of the persistent address space to processes and garbage collects the main heap.

Since the persistent store is used as the repository for all objects shared by clients, it is imperative that the contents of the store remain stable (i.e., have the ability to survive failures). This requires the use of a reliable mechanism to maintain consistency within the stable store; in our case, this mechanism is called the *stabilisation protocol.*

The store architecture used is the bi-phase Casper store described in chapter 5. However the store as described is only intended for a single user system. Here we describe additional mechanisms used to allow the store to be used in a system of distributed concurrently executing clients.

Wu and Fuchs [Wu and Fuchs 1990] describe a system whereby checkpoints are carried out on individual nodes (i.e., clients) as soon as another node requests the use of any updated data; this prevents any sharing of data modified with respect to the Stable Store Server. A major concern of their work has been to limit rollback propagation, so that the failure of any client affects only that client. Other systems [Rosenberg, Henskens et al. 1990; Henskens, Rosenberg et al. 1991] allow sharing of modified pages; in order to ensure store

consistency in such systems, the entire store must be stabilised in a single checkpoint operation.

In contrast to the system of Wu and Fuchs, clients in our architecture may share pages which have been modified with respect to the store. We adopt the solution whereby interdependent subsets of clients must stabilise together, which may occur independent of other clients in the system; these subsets are discussed in the next section. This is also in marked contrast to the approach taken by Henskens, in which all clients must stabilise together.

6.4.2. Associations

The architecture described in this chapter aims to reduce the frequency of checkpoints in any one client by maintaining a record of those clients which must be considered dependent upon one another due to the fact that they share modified pages. Only clients which are considered to be dependent on one another in this fashion need be stabilised together.

The Stable Store Server maintains information regarding the distribution and modification status of pages held by the clients; among this information is a record of which clients are dependent on each other. Dependent clients are termed *associates* and a set of mutually dependent clients is called an *association.* Each association has a corresponding *page list*, which identifies those pages modified by members of the association since their previous stabilisation; this information is used to incrementally build the associations. It is important to note that associations are dynamic in nature, with clients and pages being added and associations merging over time.

Figure 52 gives an example of how associations expand as the result of read and modification requests for various pages by different clients. Figure 52(a) corresponds to the situation where Client A has modified page x (which is shown as x' in the figure to indicate that it has been modified) and has read page z. Also, Client B has previously modified page y. Thus, the associations in the Stable Store Server record that each client belongs to a separate association and the page lists for the associations containing Clients A and B record that each has a single modified page.

Next, as shown in Figure 52(b), Client B attempts to read page x and Client C attempts to read page z. The result of these actions is shown in Figure 52(c): Client B now has access to the modified page x, Client C has access to page z. The first two associations in Figure 52(b) are now merged, since page x must now also be present in the page list corresponding to the

association containing Client B. Figure 52(d) shows that Client D then attempts to read page *y*, resulting in the state shown in Figure 52(e). The associations containing Clients B and D have now been merged into a single association because these two clients share access to the modified page *y*.

When a stabilisation is initiated by a client, only those clients belonging to the initiating client's association need be included. Thus, if Client A stabilises, then Clients B and D must also participate in this stabilisation; on the other hand, Client C may stabilise alone. All modified pages held by stabilising clients must be returned to the Stable Store Server and written back to the store as an atomic transaction. At the end of the stabilisation, the stable store will have moved into a new, consistent state. The association's page list can be used to determine which store pages are to be returned to a free page list, since up-to-date copies of those pages are stable and old versions are no longer useful. After stabilisation, the association concerned again separates into singleton sets.

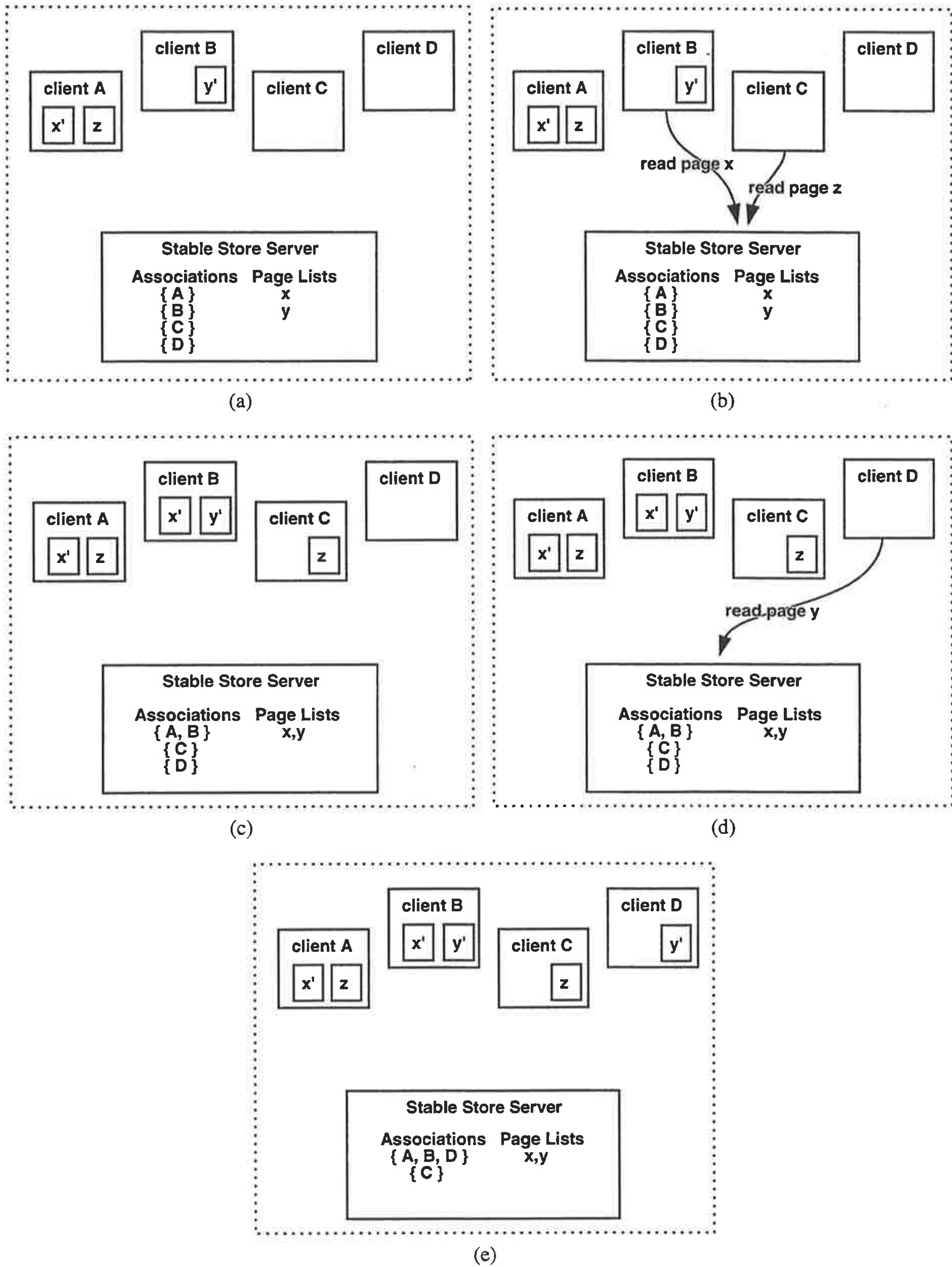


Figure 52. Associations and page lists.

6.4.3. Stabilistion

Stabilisation requires that a new consistent stable state be created from a set of pages consisting of some newly created shadow pages and some existing stable pages. Furthermore, it must be performed in such a way that it is always possible to recover the state before stabilisation, even if a failure occurs during stabilisation. A complication is that if the association stabilising its state is a subset of the entire system, care must be taken to ensure that no state information associated with clients outside the stabilising association are incorporated into the stable state created. In particular this affects the allocation of pages from the persistent address space and the stable store. Whilst being maintained within the stable store server in a manner that reflects the allocation of space by all clients, the stable state within the store must only reflect the allocations performed by the clients within the stabilising association.

The stable store server maintains two in memory versions of both the virtual address space allocation and stable store allocation maps. These are termed the *current* and *stable* versions. The current version reflects the allocations performed by or on behalf of all of the connected clients. The stable version reflects the current state of the stable store, and only differs from the stable store version during the final stages of committing the stabilisation of an association. The current and stable versions of these structures will only be the same if and when all clients stabilise together as a single association.

The stabilisation protocol is as follows: the modified pages are first written from the persistent address space to their shadow location on disk. In the normal course of events, modified pages may also be delivered to the stable store by the coherency mechanism if there is insufficient space for them within a client's physical memory. These pages are also written to their shadow locations and are regarded as having been written back as part of this stabilisation. After all modified pages have been written to their shadow sites, the remainder of the stabilisation must be synchronised with any other stabilisations which have reached the same stage. The stabilising association's page list is then traversed (recall that this holds the list of all modified pages which are to be stabilised during the current stabilisation). Pages found in the list have the appropriate shadow page allocated in the stable store allocation map, thus reflecting the allocation of shadow space for the association. Notice that the current allocation map will already hold knowledge of the allocation.

Allocation of persistent virtual memory is recorded similarly, however since a client is free to both allocate and deallocate ranges of the persistent space greater care is needed. To ensure that exactly the correct allocation is recorded in the stable structures, the stable store server maintains a log of all allocation and deallocation requests. To correctly represent the allocation of virtual memory, this log is traversed to replay these requests, and update the stable allocation map. Once updates of these allocation maps is performed they are written to their shadow location within the stable store.

The LPMap is updated during traversal of the association page list, recording the new location of data for these pages. Since the LPMap is self describing and embedded in the persistent address space, these modifications will require shadowing and this will trigger the allocation of other shadow LPMap pages recursively to reflect the creation of shadow space.

It is possible for several independent stabilisation operations to be in progress at any time since, by definition, a client can only ever belong to one association. Consequently, pages from more than one stabilisation may be written to the stable store concurrently. However, care must be taken to ensure that the stable store moves from one stable state to another in an atomic fashion. In practice, the final stages of stabilisation must therefore be serialised. In particular, modification of the LPMap, and creation of the stable allocation structures is considered an atomic action.

6.4.4. Stable Store Heap Management

The Stable Store Manager manages the stable virtual address space at the physical level; this space must also be managed at the object level. When a client requests free space, the Stable Store Manager responds by allocating unused pages. Object placement within these pages is performed by clients.

All newly allocated pages are classed as modified, since the first operation on such a page will always be a write. Consequently, they will be placed on the appropriate page lists, as described in Section 3.2. Conversely, an association's page list is cleared if that association rolls back.

From the above discussion, it would appear that the Stable Store Server understands nothing of objects or the contents of the pages it is required to supply and secure. This is not quite true, on two accounts:

- All internal information regarding the stable state of the heap of pages must be persistent.

- Garbage collection must, by necessity, be carried out at the object level.

6.5. Clients

As shown in Figure 51, a client is divided into three main threads: the PAM interpreter, the Client Request Handler and the external pager. The PAM interpreter simply executes PAM programs (compiled from Napier programs). Ideally, the interpreter should not be aware of the existence of the other parts of the client, only perceiving a single, flat, virtual address space. In reality, a few concessions must be made. The whole address range cannot be made available to the persistent heap since a small area is required within which to place both the interpreter and the coherence mechanisms. This area is reasonably small (a few megabytes) compared to the entire address space and is demand-paged by the default pager since it is not persistent.

The Client Request Handler handles all incoming messages to the client from the Stable Store Server and from other clients. The external pager handles any page faults or protection faults caused by the interpreter's attempts to access non-resident or protected pages. The Client Request Handler and the external pager jointly implement the client's part of the cache coherency protocol.

6.5.1. External Pager

The abstraction of the persistent address space within a client is managed by the external pager. The coherency protocol requires the ability to be able to detect and service page faults and to selectively protect pages and handle attempts to violate those page protections within the persistent address space. The external pager provides this functionality.

The external pager is divided into two parts: a thread which fields requests from the kernel for maintenance of the persistent address space, and a routine library which is used by the Client Request Handler to perform maintenance requests on the address space. The Client Request Handler maintains coherency. This may be as simple as changing local state information or may involve dialogue between the Client Request Handler and the Stable Store Server.

All protection exceptions and page faults caused by the interpreter's attempts to access pages are handled by the external pager. For example, when the coherency protocol requires notification of an attempt to modify a page, the page is protected against modification. Any subsequent attempt by the interpreter to modify the page will result in a page protection violation, which will be delivered to the external pager. The external pager will translate this

into a Client Modification (CM) request and forward it to the Client Request Handler. In response to coherency management requests, the Client Request Handler will call the appropriate routine in the external pager interface, which replies to the kernel; this, in turn, reschedules the interpreter. The interpreter will retry and successfully execute the instruction which originally caused the exception or page fault.

The external pager also handles the return of modified pages to the Stable Store (to relieve pressure on local physical memory). If a removed page has been modified, an up-to-date copy must be returned to the Stable Store. If a removed page has not been modified, the Stable Store Server is notified that this client no longer holds a valid page copy, so as to avoid the build-up of Export Table entries in the Stable Store Server.

However Mach 2.5, which was used to implement this system has some shortcomings. Firstly, the kernel must, as part of its memory management duties, occasionally remove pages from a user's memory cache. If the page has been modified, the kernel will return the page to the external pager. In releases of Mach derived from Mach 2.5, the Mach kernel only informs the external pager of the removal of a locally modified page. In this architecture, although the page may be unmodified with respect to the client's kernel, the page may have been modified with respect to the Stable Store by a different client. This means that the system could potentially lose the only copy of a modified page. In order to receive information on the removal of all pages (modified and unmodified), the external pager must ensure that all pages are modified (non-destructively) when they are brought into the client. This problem has been addressed in the new version of Mach (3.0) by allowing a page to be tagged as "precious", essentially being considered as modified even if no modification has occurred on that machine.

A further inconvenience is the kernel's removal of pages according to its own LRU algorithm. It would be more useful if the kernel requested the external pager to remove one or more pages, rather than sending its own choice of pages to the external pager for removal. This is due to the fact that the pages selected may contain pointers into the client's local heap area, in which case removal is a costly operation in this system, requiring copy-out of objects so referenced. The external pager can determine more appropriate candidates for efficient page removal through the available state information.

6.5.2. Atomic Access

The PAM requires that accesses to objects be atomic. This is necessary so that an object is never left in a partially modified, and hence inconsistent, state. PAM accesses are made to aligned 32-bit words, which are atomic at the machine level. However, there are some occasions when the atomicity provided at this level is insufficient. Such cases include accessing multiple word objects, such as real numbers (which are 64-bit quantities), bit-maps and discriminated unions. A discriminated union is represented by a pointer to the data and a tag field, and these may be stored non-contiguously. This requires a mechanism capable of providing the interpreter with atomic access to multiple data locations at arbitrary addresses. This may be achieved using a structure which we call a *latch*. The semantics of a latch is analogous to a door latch: it may be set before the door is closed, but once the door is closed, the door will not open again until the latch is released.

Atomic access to multiple locations at arbitrary addresses may be implemented via latching each affected page. Two latches are provided per page – a read latch and a write latch. A latch, when set, prevents the release of the page to any other client for the purpose indicated by the kind of latch. If a page is required for an atomic read operation, the write latch is set and so a write operation by another client occurring part way through the read operation is prevented. If an atomic write is desired, the read latch is set to prevent the page from becoming shared part way through the write. The design of the latching mechanism has aimed for efficiency, particularly for common cases, such as when only one page is needed and the page is already resident.

Latching is implemented in each client through a data structure called the *latch table*. The latch table is a fixed size array of address-range, latch type pairs. The architecture of the PAM is such that at most four separate address ranges need ever be latched for the most complex atomic action. Thus the latch table has four entries. Address ranges are represented as either an individual page id, or a page id and a number of pages.

An instruction that requires atomic access to ranges of memory first places the address ranges into the latch table, this requires that it know in advance all of the address ranges required, again this is possible in the PAM interpreter. Once all the address ranges are loaded a global flag is set to inform the client FSA that a latched operation is in progress. Then the pages described in the latch table are loaded. This is achieved by simply touching each page. The client FSA will load any pages not already present and will prevent the loss of any pages

that are present. Once all pages have been touched the atomic operation may proceed. Once it has completed the latch table is tagged as empty (by setting the number of entries counter to zero) and the global flag is unset.

When the need arises to access more than one page, pages are latched serially and in a defined order (in fact, ascending address order) to prevent circular dependencies with competing clients, and hence avoid deadlock. A further rule is required to prevent deadlock if a competing client already has some of the pages needed to complete an atomic operation latched itself. If a client has some pages latched but does not have the complete set to allow it to proceed, it will release the latched pages if the pages it does not yet have latched are of lower address. The combination of defined order of acquisition and a pre-emption rule prevent deadlock.

The need for the more expensive mutex locks is obviated, since latches are:

- only ever set by the interpreter,
- released by the interpreter or by the Client Request Handler only when the interpreter is guaranteed to be blocked, and
- only read by the Client Request Handler.

6.5.3. Local Heap Management

Each PAM interpreter executing in a client maintains a *local heap* for local object creation; this is a previously unused set of contiguous persistent pages. Local heaps are small enough to always remain resident within the client's page cache during normal execution. Greater locality of reference may be obtained in relation to other systems that do not use the local heap model. This can result in improved performance from better page fault behaviour and improved processor cache utilisation.

Local heaps may be independently garbage collected; to support such garbage collection, all references into a local heap must be retained within the client. Any page containing references into a local heap is not exported from the client until the referenced objects have been removed from the local heap.

If transient objects are confined to a localised area, they may be distinguished from persistent objects resident in the total persistent address space and hence may be garbage collected locally at low cost. Local heaps may be safely garbage collected provided that no external references (from other processes or the Stable Store Server) point into them.

Fortunately, the creation and export of such pointers is easily detected, making this technique tractable; these matters were covered in detail in chapter 2.

The implementation permits, and requires, a single external pointer into each local heap. This pointer is to the *process header* representing the process executing in the local heap and is used as a root for local garbage collection. Process headers are reachable from the root of persistence, making it possible to restart processes in the event of a failure. All objects created locally are reachable from the process header at the time of their creation, because of the manner in which stack frames are organised by the interpreter.

6.6. Cache coherency

The introduction of multiple clients which can concurrently access pages within the Stable Store by operating over a local cache poses a major challenge for the design of a cache coherency protocol. The bus-based multiprocessor hardware cache coherency methods (e.g., write-through, copy-back, etc.) [Archibald and Baer 1986] are inappropriate. For example, the clients may be operating over a distributed network, where the communication costs are greater than in a bus-based multiprocessor (especially for operations such as broadcasting and snooping).

The coherency protocol has been specified as two finite state automata (FSA), representing the states of a page within each of the clients and the Stable Store Server. Client-server interaction is modelled by transition interactions between the two automata. Client-client interaction is represented within the client's automaton. In the implementation, these FSA are used as the basis for the code which maintains page coherency in the entire system.

The general aim of the protocol is to allow multiple clients to read the most up-to-date copy of a page, or a single client to write to a page without compromising the coherency of the pages. All read and write requests are made directly to the Stable Store Server. Depending upon the state of a page, read requests arriving at the Stable Store Server may be forwarded to a client with an up-to-date copy of the page or the Stable Store Server may service the requests directly. The aim is to maximise the freedom with which a client process is able to run, and to prevent the Stable Store Server from becoming a bottle-neck for page retrieval and supply.

6.6.1. Client Finite State Automaton

A simplified version of the Client FSA is shown in Figure 53. This automaton represents the set of states to which a page may belong while it is held in a client's cache. The transition of a page from one state to another is initiated by the client's receipt of a message referring to that page, from either the Stable Store Server, another client or the client's external pager (due to the interpreter's attempt to access the page). The states and signals shown on the diagram are explained in detail in Appendices A and B, respectively. Some aspects of the FSA in Figure 53 will now be explained; it should be emphasised that this explanation will not attempt to cover all the states of the FSA.

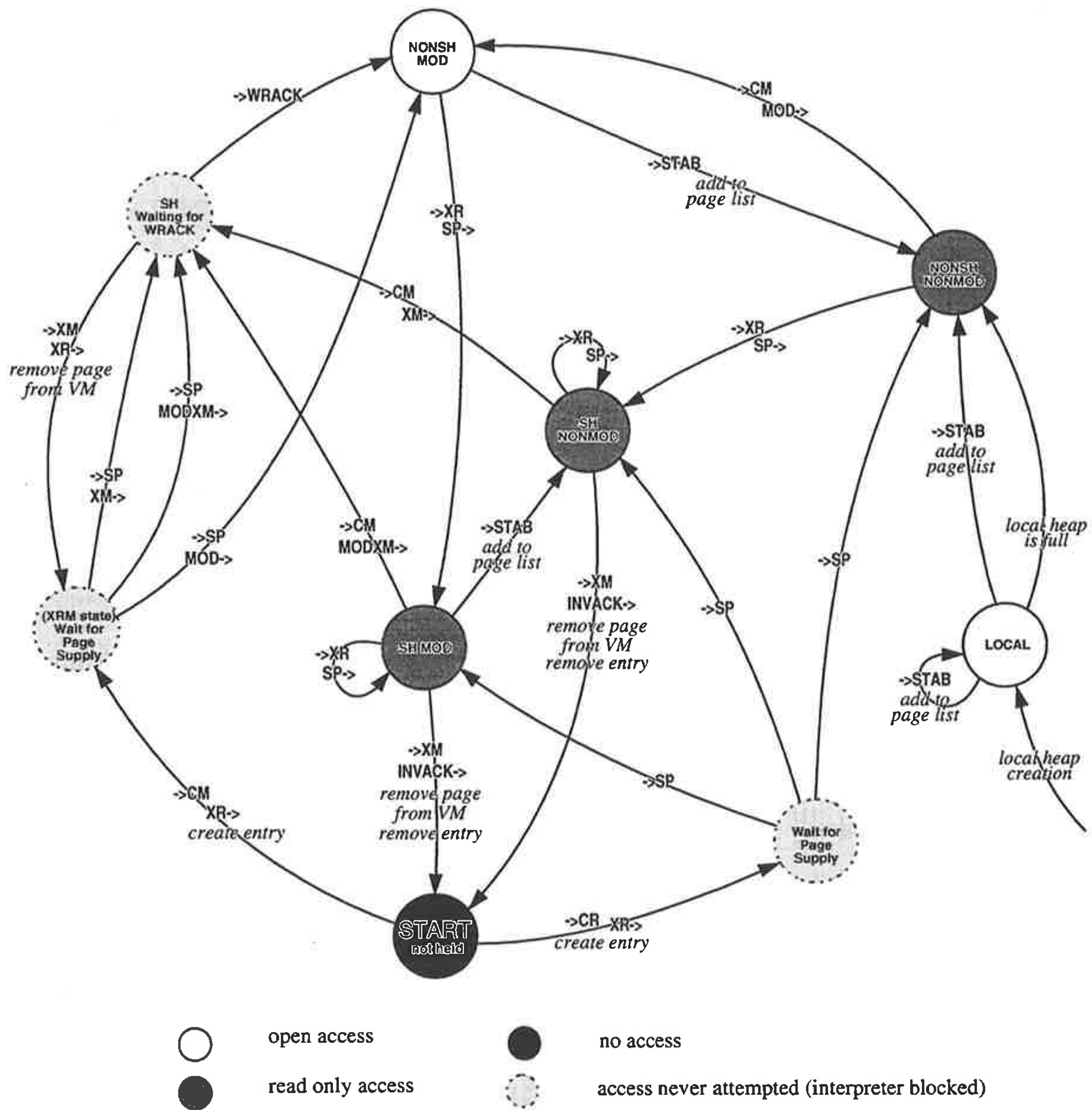


Figure 53. Simplified Client FSA.

The major points to note in relation to the operation of the Client FSA are the following:

- A page is considered to be modified if it has changed since it was last stabilised, even if it was not the client currently holding the page that performed the modification. If a modified page copy is held in the stable store, it is still classed as modified until it is checkpointed during a stabilisation.
- When a page is shared, a modifying client must gain exclusive access to the page before proceeding. It therefore requests modification permission from the Stable Store Server and awaits the latter's acknowledgement before proceeding with the modification.
- First-time modification of a non-shared page only requires the forwarding of a modification (MOD) signal to the Stable Store Server. The client need not wait for the Stable Store Server's acknowledgement before performing the modification on such a page. This optimisation allows asynchrony, as the interpreter can continue execution while the modification is still in transit to the Stable Store Server.
- A page returns to a non-modified state when it is stabilised; at this time, any copies held by clients are identical to the stable copy held in the stable store.

To give some appreciation of the client-related part of the coherency protocol, we illustrate some of the more interesting aspects of the protocol below. Figure 54 shows the interpreter of Client A attempting to read page *z*; the request goes to the page cache, where it is discovered that the page is not resident. All non-resident pages are implicitly in the 'START' state. As described earlier, the external pager will field the page fault on page *z* and forward a client read (CR) request to the Client Request Handler.

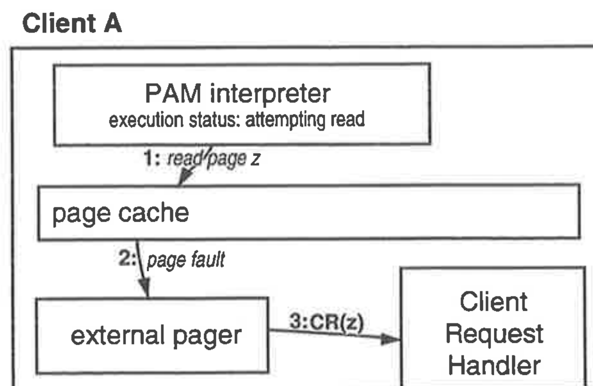


Figure 54. Client A attempting to read a non-resident page.

The Client Request Handler receives the request, and forwards an external read (XR) request to the Stable Store Server, as shown in Figure 55. It also creates an entry for the page in the

local Import Table and records the page's state (Wait for Page Supply) in this table. At this time, the interpreter is blocked pending the delivery of the page.

The page may arrive in the client under a number of different circumstances. If the page was exported from the store since it was last stabilised, the Stable Store Server may forward the read request to one of the clients which hold a valid copy of the page; this client will, in turn, forward a copy of the page to the requesting client. In the case where the Stable Store Server holds an up-to-date copy of the page, the page is sent directly from the server to the client.

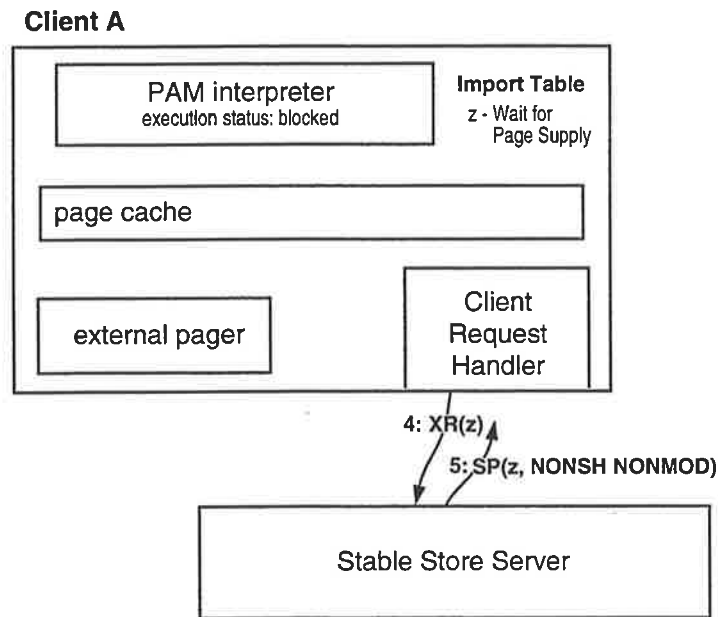


Figure 55. Client A forwarding an XR request to the Stable Store Server.

Assuming that the page is not held by any other client, the Stable Store Server will send the page copy directly to Client A using a supply page (SP) signal and the page will arrive tagged for use as a non-shared, non-modified page. Upon receipt of the page, the state of page *z* is updated in the Import Table by the Client Request Handler. This process is shown in Figure 56 below. Once the page arrives, the Client Request Handler uses the external pager to place the page in the client's cache; it is protected for read only access, because the page is non-modified and the Stable Store Server must be informed of any modifications to the page. The interpreter then continues execution.

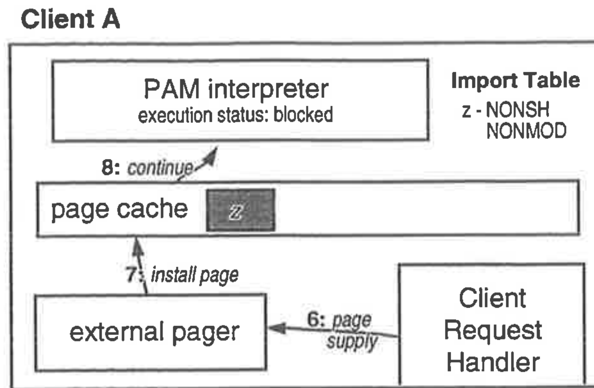


Figure 56. Once the page has been supplied by the Stable Store Server.

Pages in the non-shared, non-modified state may be freely read by the PAM interpreter. However, if the interpreter attempts to modify such a page, another exception will be raised. The external pager again fields the access exception and forwards a client modify (CM) signal to the Client Request Handler. Figure 57 illustrates the attempt to modify page *z*.

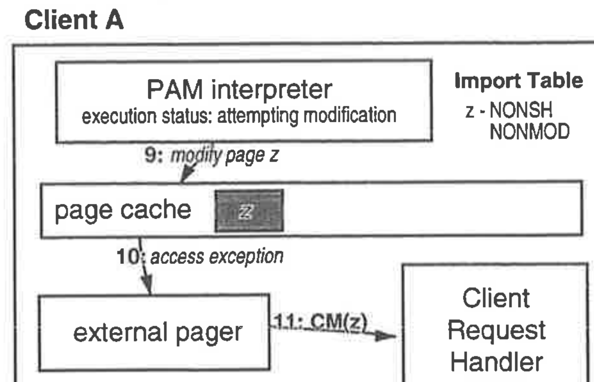


Figure 57. Client A attempting to modify the page.

As a result of the page exception in Client A, three things happen: first, the page is opened for write access; second, it is moved into the 'NONSH MOD' state; finally, a modification (MOD) signal is forwarded to the Stable Store Server. The interpreter may resume execution once these tasks have been performed, as shown in Figure 58. The Stable Store Server, upon receipt of the MOD signal, will allocate a shadow page for this page. Furthermore, the modification of this page may result in new interdependencies between clients and hence expansion of associations. Under ordinary circumstances, the associations are kept up-to-date as modifications proceed to pages in the system.

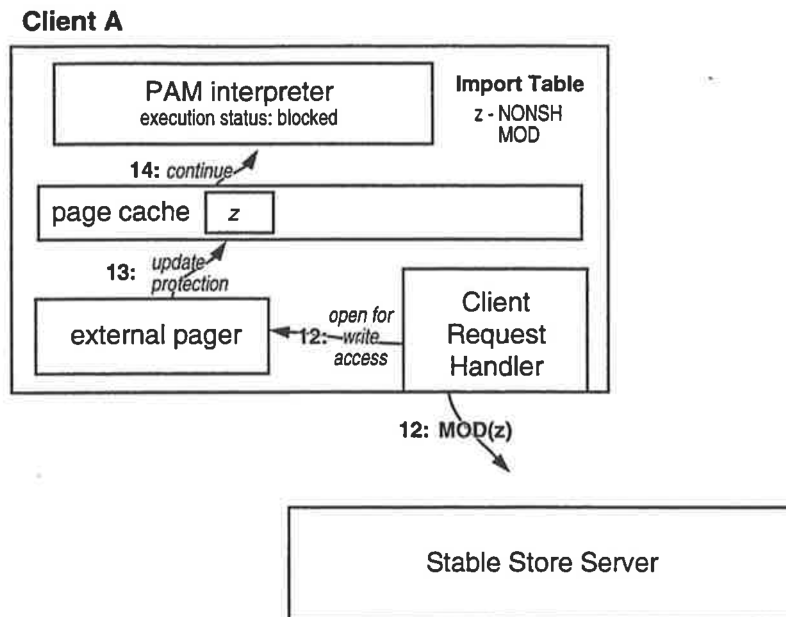


Figure 58. The MOD request being sent to the Stable Store Server.

Complications may arise if page z is being manipulated by other clients while the MOD signal is still in transit. If two other clients (Clients B and C) request a copy of page z, those requests may arrive in the server ahead of Client A's MOD signal; this situation is shown in Figure 59.

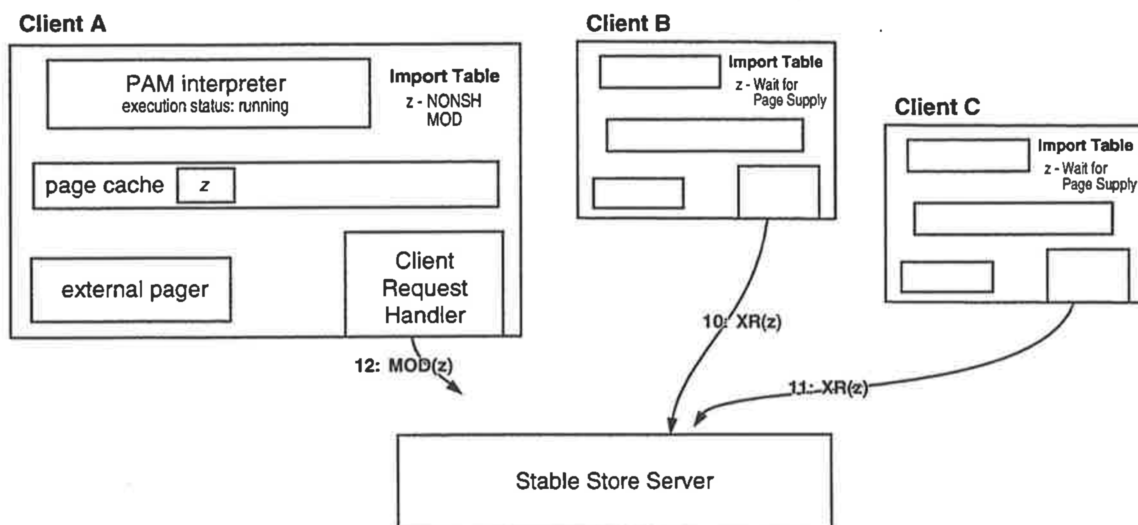


Figure 59. Two other clients request copies of page z.

Upon receipt of Client B's XR request, the server will forward an XR signal to Client A, indicating that an up-to-date copy of page z must be sent to Client B; this is depicted in Figure 60. The XR request from Client C (recall Figure 59) is then received by the Stable Store Server. At this time, the signal could be forwarded to either Client A or Client B; in this example, the XR is again forwarded to Client A, as shown in Figure 60. When each of these XR signals are received by Client A, a copy of page z is made and forwarded, as part

of an SP signal, to the requesting client. Once these SP signals have reached their destinations, Clients A, B and C will hold Client A's version of page z in the shared, modified state. If the MOD signal from Client A has not arrived in the Stable Store Server, the server will not be aware that page z is modified. Therefore, the associations containing Clients B and C will not have been merged with that containing Client A.

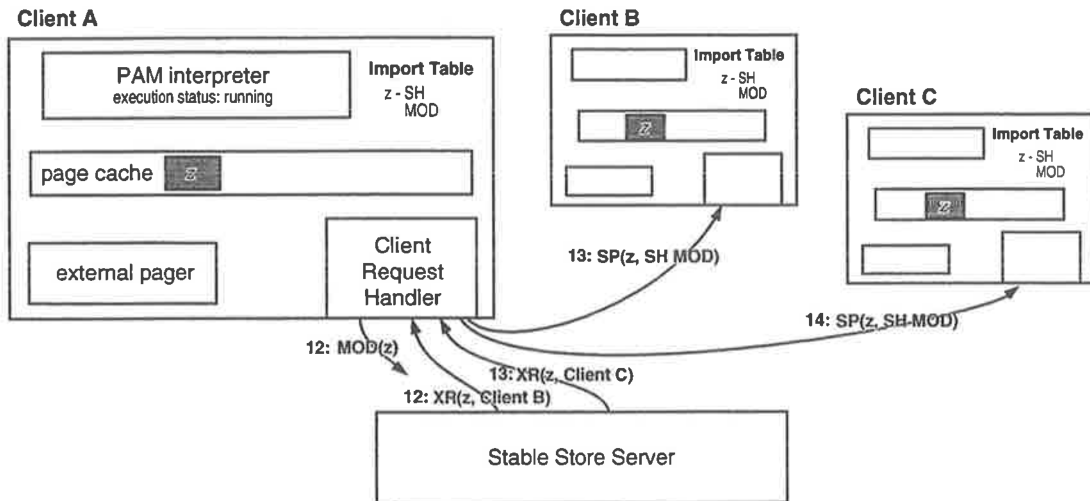


Figure 60. Clients B and C receive copies of page z.

Should Client B then attempt to modify page z, a MODXM (external modification of a previously modified page) signal must be sent to the Stable Store Server. This indicates to the server that Client B requires modification permission for a page which has already been modified with respect to the stable store copy. Upon receipt of the MODXM, the Stable Store Server will consult its state information for the page; in this case, the state of page z indicates that it is unmodified. The Stable Store Server is aware, at this time, that a MOD signal is in transit from the modifying client.

Since continuing to service the MODXM will result in a depletion of the record of clients currently holding a page, and since all the clients currently in this recorded list share a modified page copy and are therefore interdependent on one another, the associations containing these clients must be merged. The server can, therefore, correctly maintain the associations of all clients holding a page, despite the unpredictable ordering of the arrival of signals. The MODXM signal is then treated as a normal XM signal and write acknowledgement will eventually be sent to the original requesting client (Client B in this case).

From the latter part of the above example, it should be clear that there are a number of problems introduced when the interaction between multiple clients is considered. These interactions account for the remainder of the simplified diagram in Figure 53, which will not

be discussed further here. An attempt has only been made to illustrate a small section of this diagram and to mention some of the complications relating to that section.

6.6.2. Stable Store Server Finite State Automaton

The FSA for the Stable Store Server is shown in a simplified form in Figure 61. It describes the various states to which a page held in the stable store may belong. Any page which has not been exported from the Stable Store Server implicitly belongs to the 'START' state. Transitions are initiated by requests from the connected clients. Appendices C and D give a full description of the states and signals used in the Server FSA. As with the preceding section and its discussion of the Client FSA, only some aspects of Figure 61 will be discussed here.

Figure 61. Simplified Server FSA.

Information regarding the distribution and modification of pages is maintained by the Stable Store Server on a per-page basis. A data structure known as the *valid list* (V-list) contains the identity of all clients holding a valid copy of a page. Similarly, the *dependency list* (D-list) records which clients hold or have held a modified copy of a page since the last stabilisation. The Server FSA instigates all updates to the pages' V-lists and D-lists. Information on the shared state of pages is therefore always up-to-date in the Stable Store Server; however, a page may have been modified by a client before the Stable Store Server is aware of the modification. In such a case, a MOD signal will have been sent to the Stable Store Server by

the modifying client. Upon receipt of a read request, the Stable Store Server either provides the client with a copy of the required page, if it has a reliable copy available, or it consults the page's V-list to select another client capable of forwarding an up-to-date copy of the page directly to the requesting client.

The operation of the Server FSA will now be illustrated by example. Consider the situation in Figure 62, showing the modification of a shared page; a number of clients have requested and received copies of a particular page (page x). Assuming that the clients have not previously attempted to modify the page, it will be in the shared non-modified state in both the Stable Store Server and the clients. If one client (Client A) attempts to modify the page, an external modification (XM) signal will be sent to the Stable Store Server and this client will await write acknowledgement.

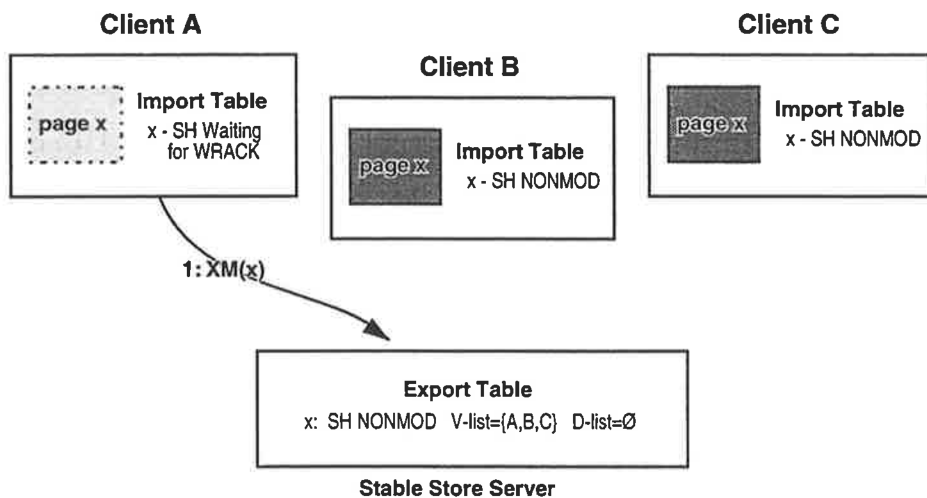


Figure 62. The page x is shared among several clients.

Upon receipt of the XM signal by the Stable Store Server, all clients belonging to the page's V-list, except the requesting client, must invalidate their copies of the page. Invalidating XM signals are sent to those clients by the server as shown in Figure 63. Once these signals have been sent, the page's state in the server will change to the 'Wait for INVACK XRq=Ø' state, indicating that the Stable Store Server is awaiting invalidation acknowledgement (INVACK) signals from the notified clients ('Wait for INVACK') and no further read requests have been received for the page (i.e., the XR queue is empty, indicated by 'XRq=Ø').

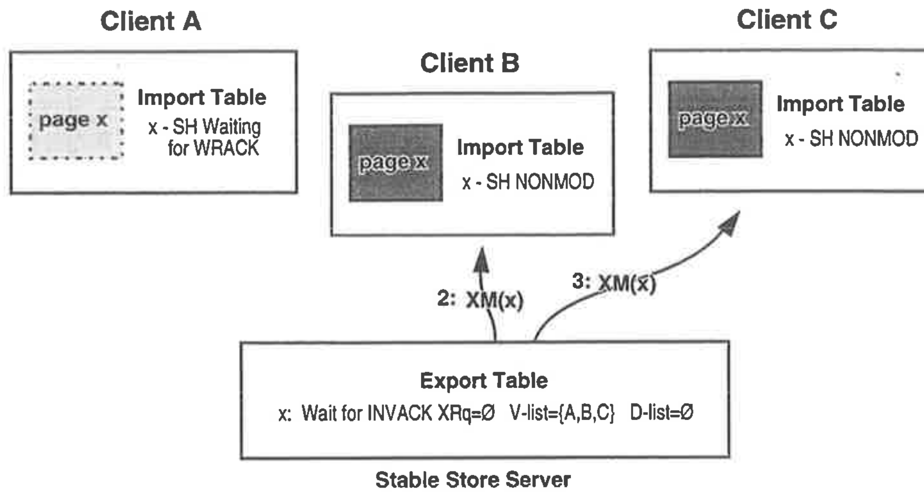


Figure 63. Invalidating XM signals being sent to clients.

Clients B and C will invalidate their copies of page *x* as a result of receiving the XM signals from the Stable Store Server. Records kept for the page will be deleted from their Import Tables and the page will implicitly move back to the START state in each of these clients. INVACK signals will then be sent to inform the server of the completion of the clients' page invalidations. The result of the invalidations is shown in Figure 64.

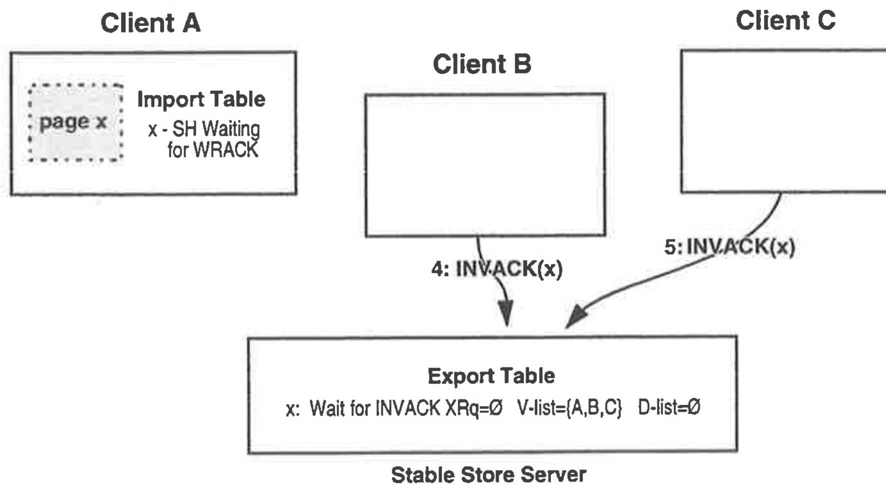


Figure 64. Clients B and C responding to the invalidating XM signals.

As each INVACK signal is received by the Stable Store Server, the corresponding clients are removed from the V-list. This process continues until there is only one remaining member in the V-list, the client which originally requested modification permission, as depicted in Figure 65. A write acknowledgement (WRACK) signal is then forwarded to Client A, and the page moves into the non-shared, modified state. As soon as the WRACK signal has been sent to Client A, the client will be added to the D-list for page *x* recording Client A's dependency on the modified page. Upon receipt of the WRACK signal by Client A, page *x* will move into the non-shared, modified state and the interpreter will resume execution.

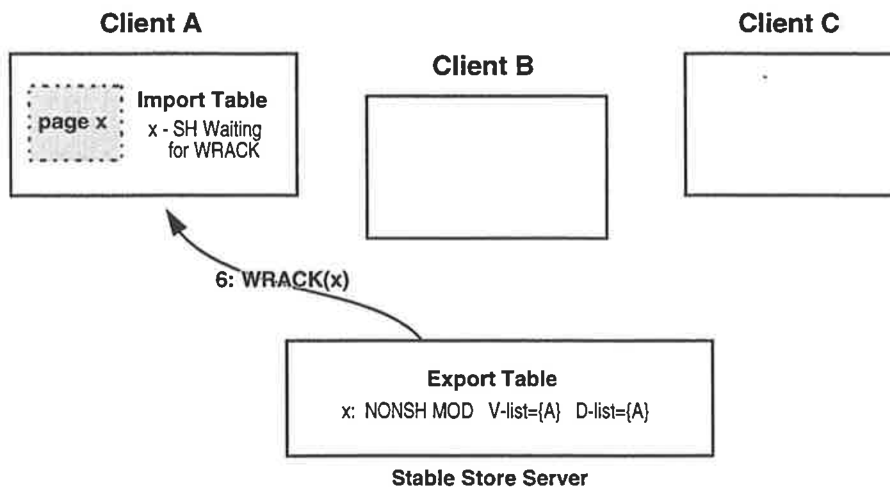


Figure 65. A WRACK signal being sent to Client A.

In a realistic environment, clients are likely to compete for access to the same page. Consider a scenario where Client C also attempts to modify page x. This scenario is illustrated in Figure 66, which replaces Figure 63 in the above sequence. If the XM request from Client A reaches the Stable Store Server before the XM from Client C (recall Figure 62), Client A's request will receive preference.

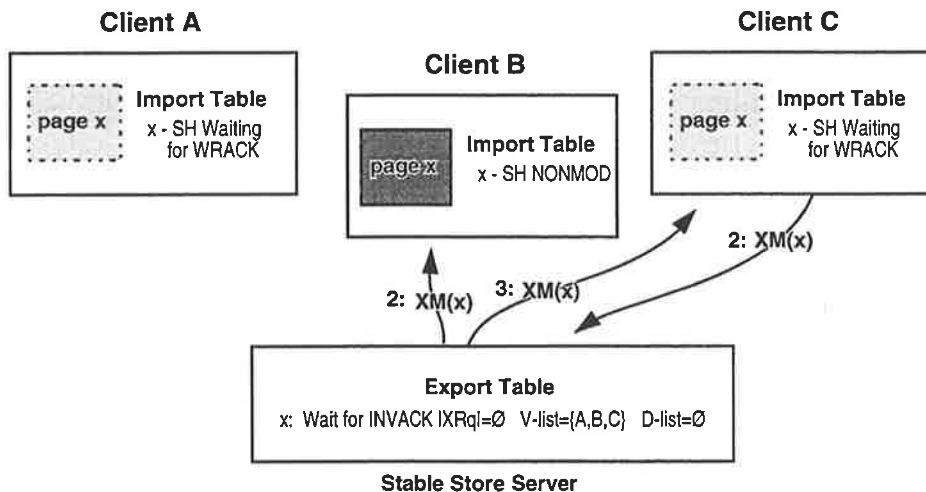


Figure 66. Client C also requesting modification permission via an XM signal.

Upon receipt of the modification request from Client C, the Stable Store Server is aware that the client's interpreter is halted and that an invalidating XM signal has been sent to Client C. The XM request can therefore be treated as an invalidation acknowledgement signal and Client C is removed from the page's V-list. This update is illustrated in Figure 67.

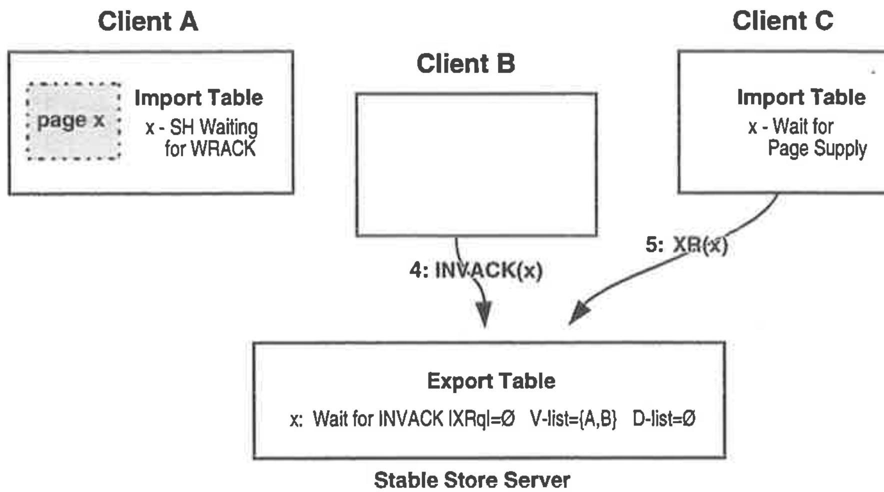


Figure 67. Client C's XM has been treated as an INVACK signal.

Since Client C's XM signal has been treated as an INVACK, the client need not respond to the incoming XM with an actual INVACK signal. Consequently, when Client C receives the invalidating XM from the server, it must simply invalidate its page copy. Since the interpreter in Client C is blocked on an attempt to modify page *x*, the Client Request Handler will send a request to the server (using an XR signal) for another copy of the page. This is shown in Figure 67 and will result in Client C receiving the up-to-date version of the page after the original modifying client (Client A) has received write permission, as shown in Figure 68, and completed its modification.

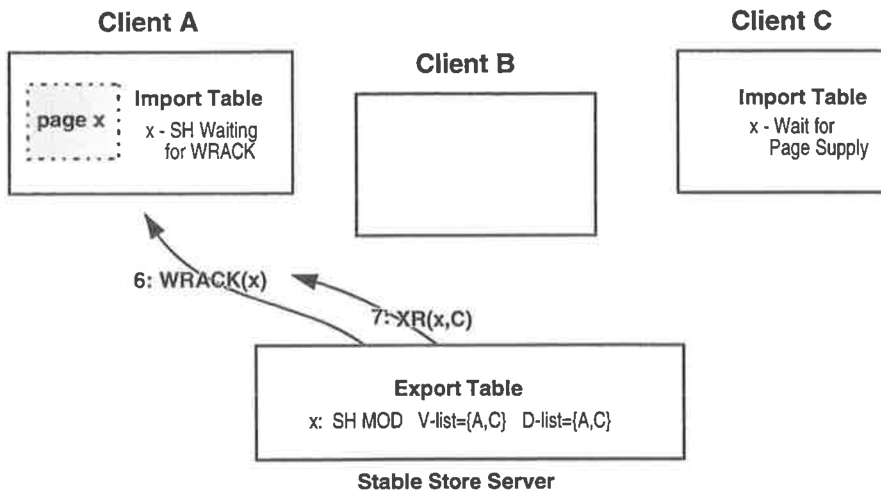


Figure 68. Client C's XR being forwarded to Client A after the WRACK signal.

6.6.3. Interaction between the Automata

Section 6.6.1 described the operation of the Client FSA; the Server FSA was covered in Section 6.6.2. In the latter section, some interaction between the two FSA was shown, since the states of pages within clients were given as the behaviour of the Server FSA was explained. In this section, the interaction between the Client FSA and the Server FSA will be

further illustrated, to emphasise the manner in which both contribute to the maintenance of coherency of the stable store.

A particular scenario, again involving three clients, will now be described. The first snapshot, in Figure 69, indicates that the three clients have sent XR requests for the page *p* to the Stable Store Server. The first request to arrive is that from Client A and this client is placed in the page's V-list. The Stable Store Server has not previously exported a copy of the page to any client; therefore, it can service the first request itself by forwarding a copy of the page to Client A. Before the remaining two read requests arrive at the Stable Store Server, page *p* is in the state 'NONSH NONMOD'. Client A, upon receipt of the SP signal, moves its copy of the page into the 'NONSH NONMOD' state; this is the situation shown in Figure 69. Clients B and C have entries for the page in their respective Import Tables which indicate the state of the page to be 'Wait for Page Supply'.

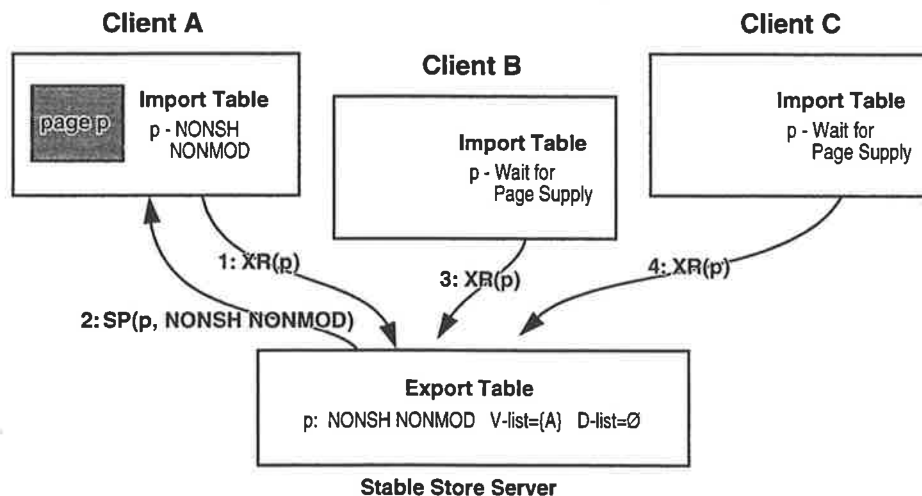


Figure 69. Multiple clients requesting copies of page *p*.

Next, the other two XR signals are received by the Stable Store Server. As a result, these clients are added to the page's V-list, yielding the V-list shown in Figure 70. When the first of these signals is received from Client B, the page moves to the 'SH, NONMOD' state in the Stable Store Server and the XR signal is forwarded to Client A. Client A makes a copy of the page, changes the page's state to 'SH NONMOD' and forwards the copy to Client B via an SP signal. Client C's request, when received by the Stable Store Server, may be forwarded to either Client A or Client B. An arbitrary choice is made in an attempt to reduce a potential bottleneck of page requests at a client. As can be seen from Figure 70, Client B was selected to supply page *p* to Client C in this case and the XR is forwarded to Client B: upon receipt of the XR, Client B makes a copy of page *p* and sends it to Client C via an SP signal.

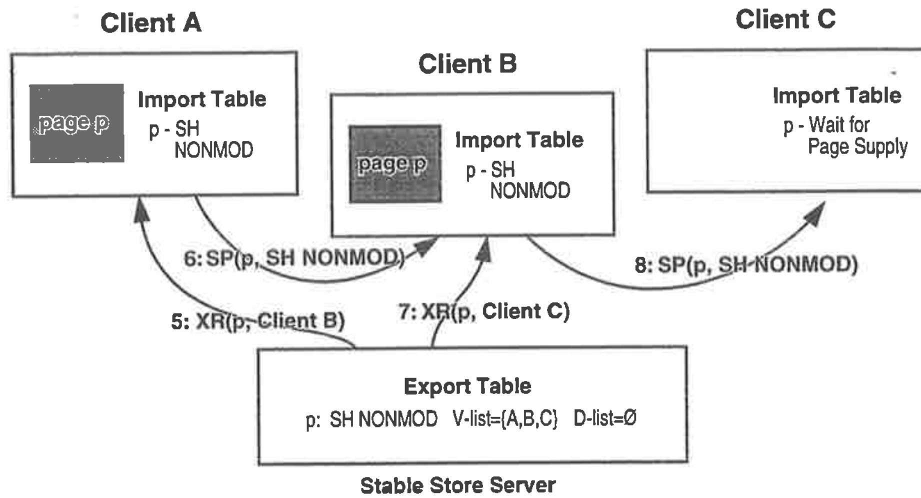


Figure 70. Servicing the read requests from Clients B and C.

When the SP signal is received by Client C, it changes the state of the page to that arriving in the signal; this is depicted in Figure 71. Client A then sends an XM request to the Stable Store Server as a result of this client's interpreter attempting to modify the write-protected page *p*. This page has moved to state 'SH Waiting for WRACK' in Client A, in preparation for the arrival of the expected WRACK signal. Upon receipt of the XM signal, the Stable Store Server must traverse the list of clients which hold current copies of the page and request those clients to invalidate their copies; this is achieved by sending XM signals to the clients concerned. The page moves into the 'Wait for INVACK XRq=0' state in the Stable Store Server.

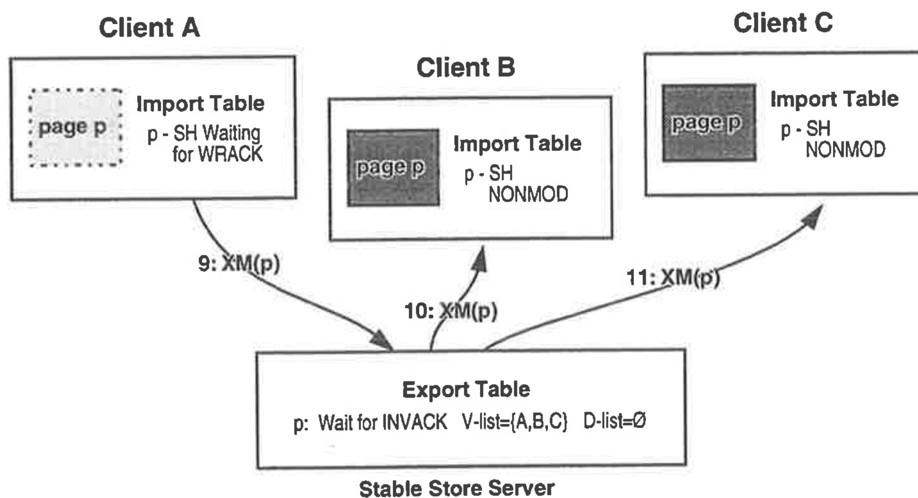


Figure 71. Client A's modification request being serviced by the Stable Store Server.

The result of these invalidating XM signals is shown in Figure 72, where the Clients B and C have removed the copies of page *p* from their local caches, along with any internal records held for the page. These clients then reply to the Stable Store Server with INVACK signals. The page moves implicitly into the 'START' state in each of Clients B and C.

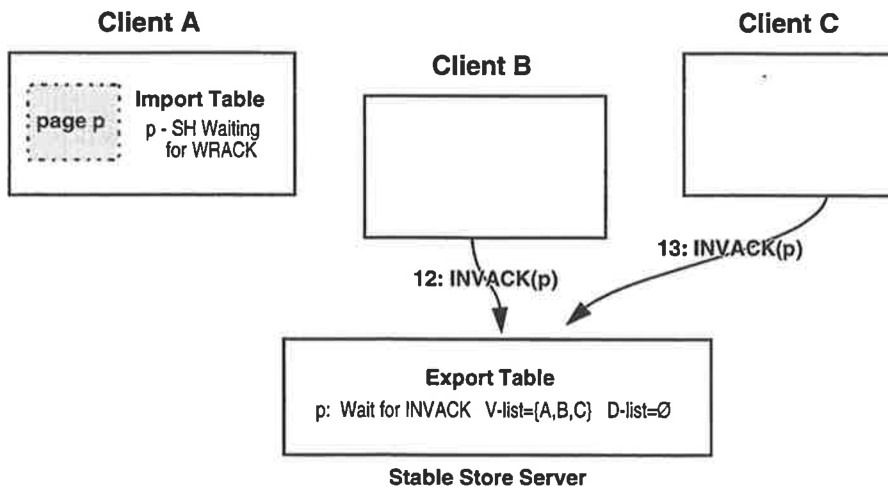


Figure 72. The invalidating clients acknowledging their invalidations of page *p*.

Upon receipt of the INVACK signals by the Stable Store Server, Clients B and C are removed from the V-list for page *p*; this is shown in Figure 73. Once the V-list is reduced to a single member (the original requesting client), that client is added to the page's D-list. A WRACK signal for page p is now sent to Client A and the page is moved to the 'NONSH, MOD' state in the Stable Store Server. Upon the arrival of the WRACK signal in Client A, the page is moved to the 'NONSH MOD' state in that client.

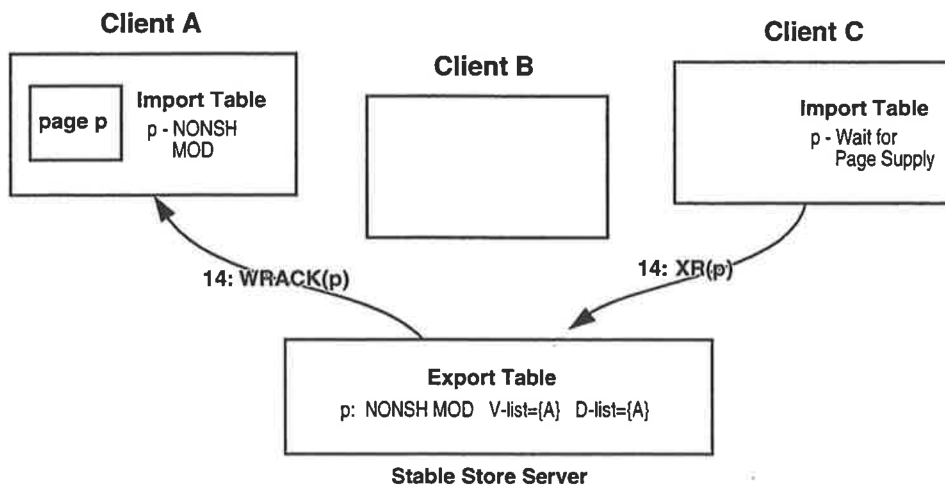


Figure 73. Write acknowledgement being forwarded to Client A as Client C requests another copy of page *p*.

At the same time as the Stable Store Server is sending the WRACK signal to Client A, Client C sends an XR request for page *p* to the Stable Store Server. As shown in Figure 74, this request causes Client C to be added to both the V-list and D-list for page *p* in the Stable Store Server and the request to be forwarded to Client A. Another consequence of the XR request is that the page is moved into the 'SH, MOD' state in the Stable Store Server. Client A, upon receipt of the XR signal, moves the page into the 'SH MOD' state and forwards a copy of

the page to Client C via an SP signal. When this page is received, Client C will also hold the page in the 'SH MOD' state.

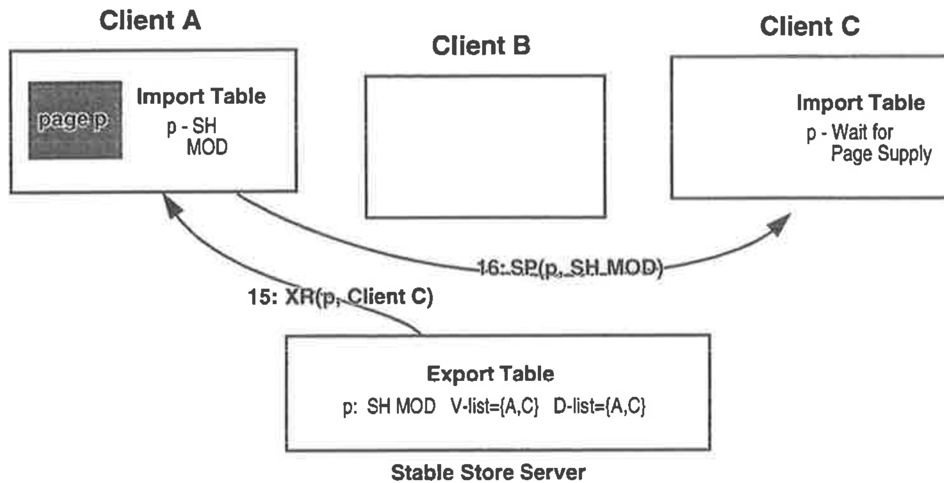


Figure 74. Client C's read request being serviced by the Stable Store Server and Client A.

6.6.4. Omissions from the Simplified Automata

As already mentioned, the automata shown in Figures 53 and 61 are simplified versions of the FSA used to implement the system. These simplified automata have been employed in order to simplify the description of the essentials of the coherency management in the system. At this stage, however, it is worth characterising the principal omissions which have been made.

In the Client FSA, the main omissions are:

(C1) XR requests which arrive from other clients, via the Stable Store Server, before the client has received its copy of a page.

(C2) Invalidating XM requests arriving from the Stable Store Server ahead of SP signals from another client.

(C3) As discussed in Chapter 2, copy-out space is required to allow locally created objects to be garbage collected independently of overall stable store garbage collection. The potential problem of copy-out space being unavailable when a read request arrives for a page which contains pointers into the local heap.

(C4) The handling of an XR request forwarded from the Stable Store Server, and arriving in a client after that client has sent an XM signal to the Stable Store Server for the page.

(C5) The complications introduced through kernel-initiated removal of a page from a client's local cache, as part of normal virtual memory management.

As for the Server FSA, the following are the principal omissions:

(S1) Pages are never returned by clients – this follows from (C5) above.

(S2) Following from (S1), no indication is given of how to handle the receipt of unexpected INVACK signals due to page removal signals being treated as INVACK signals.

6.7. Conclusions

We have described in some detail the design of a single server distributed persistent system. Of particular interest is the associations mechanism which tracks causal links between separate programs executing on separate clients, ensuring that a self consistent state is always represented within the stable store. This mechanism is a special case of the more general problem of tracking causal relationships and generating consistent views of computation in a distributed environment. This problem is the subject of the next chapter.

The Casper system has verified the utility of the external pager mechanism for the provision of both conventional distributed shared memory, and for the creation of a persistent virtual address space. The system could potentially have been implemented above a conventional operating system such as Unix, using access protection exceptions to trigger the coherency system, however the system would continually have been in conflict with the operating systems page swapping operations and would be required to carry out unnecessary copy operations as pages were transferred. The success with the external pager in the Casper system has been a major influence in the design of the container manager abstraction in the Grasshopper operating system. This system is described in Chapter 8.

Chapter 7. Grasshopper

7.1. Introduction

In the preceding chapters we have described tactics for exploiting page based hardware to support orthogonal persistence. These tactics have often been compromised by the inappropriate design of the host operating systems. This last chapter describes the design of aspects of a new operating system (called Grasshopper) which supports orthogonal persistence as an intrinsic feature. This new operating system includes specific support intended to make the use of page based persistence mechanisms natural to implement. Grasshopper is a joint project between the Universities of Sydney and Adelaide in Australia and the University of Stirling, Scotland.

This chapter begins with an examination of why a new operating system has been built, why current systems fall short, and some of the goals of the new design. A brief overview of the basic abstractions and computational model of the system is presented.

We have seen in Chapter 6 how tracking causal interdependencies in the Casper system can be exploited to reduce the costs of melding the volatile state with the stable store. In this chapter we survey the general basis for tracking and characterising causal links and how more general mechanisms can be constructed to allow consistent system states to be captured without requiring global synchronisation. The major thrust of the remainder of the chapter is to describe the underlying mechanisms by which the Grasshopper operating system allows individual parts of the system to maintain stable state, using locally optimal mechanisms but coordinated to ensure global reliability.

7.2. Why a new operating system?

Tanenbaum [Tanenbaum 1987] listed the four major components of an operating system as being memory management, file system, input-output and process management. The nature of these four components is different in persistent systems. In a persistent system, the functionality of the file system and memory management are replaced by the persistent store. In many operating systems, input-output is presented using the same abstractions as the file system; clearly this is not appropriate in a persistent environment. Some persistent systems require that the state of a process persists; this is not easily supported using conventional operating systems. It is therefore to be expected that an operating system designed to support persistence will have a different structure from a conventional operating system and will provide a different set of facilities.

In previous chapters we have described attempts to manufacture persistent environments using conventional operating systems. All of these have been compromised to some degree by the mismatch between the abstractions presented by the operating system and the goals of orthogonal persistence. In all of these systems the implementor expends considerable effort manufacturing a virtual machine abstraction on top of the model presented by the host operating system. On top of this new virtual machine the persistent system is implemented. Such a tactic produces systems with at least two superfluous layers of abstraction and additional opportunity for errors.

We can summarise the principal requirements of a persistent operating system as follows [Dearle, Rosenberg et al. 1992]:

i. The major requirement is support for persistent objects as the basic abstraction.

ii. These objects must be both stable and resilient. The system must reliably manage the transition between long and short term memory transparently to the programmer.

iii. Processes must be integrated with the object space in such a way that process state is itself contained within persistent objects. Thus processes themselves become resilient.

iv. Although the persistent store is uniform, there is still a requirement to be able to restrict access to objects for the same reasons that file systems provide access control mechanisms. Any operating system supporting persistence must therefore provide some protection mechanism.

7.3. Why conventional Hardware?

Grasshopper is intended to run on conventional hardware architectures. In the previous chapters we have examined tactics for, and implementations of, persistent systems that exploit conventional page based hardware architectures. Grasshopper has been designed in the belief that these tactics along with an operating system design specifically intended to support them, will result in a system with considerable performance and useability benefits. Other benefits of a conventional hardware approach are:

- The performance of these systems is increasing dramatically every year due to competition between, and massive investment by, the hardware vendors.
- These architectures are highly available. It is easy to disseminate research results by providing copies of the system to interested parties.

- Should commercialisation become a possibility, a totally software implementation is easier to market than a solution including specialised hardware

Using conventional hardware, and in particular exploiting the page based virtual memory paradigm, has proven to be very successful in the designs studied in the earlier chapters. The major problem encountered with these systems has been the poor correspondence between the abstractions for control of virtual memory provided by existing operating systems, and the demands made in implementing orthogonal persistence. One of the clear design goals of the Grasshopper system is the provision of a suitable abstraction.

7.4. Grasshopper

Grasshopper relies upon three powerful and orthogonal abstractions: *containers*, *loci* and *capabilities*. Containers provide the only abstraction over storage, loci are the agents of change (processes/threads), and capabilities are the means of access and protection in the system.

Conceptually, loci execute within a single container, their *host container*. The data stored in a container is supplied by a *manager*. Managers are responsible for maintaining a consistent and recoverable stable copy of the data represented by the container. As such, they are vital to the removal of the distinction between persistent and volatile storage, and hence a cornerstone of the persistent architecture. It is the operation of the managers and their inter-relationship with the kernel in the maintenance of a consistent recoverable system state that is the major topic of this chapter.

7.4.1. Containers

In systems which support orthogonal persistence the programmer does not perceive any difference between data held in RAM and that on backing store. This idea leads naturally to a model in which there is a single abstraction over all storage.

Grasshopper adopts this model by implementing regions called *containers*. Containers are the only storage abstraction provided by Grasshopper; they are persistent entities which replace both address spaces and file systems. In most operating systems, the notion of a virtual address space is associated with an ephemeral entity, often termed a *process*, which mutates data within that address space. In contrast, containers and loci are orthogonal concepts. A Grasshopper system consists of a number of containers, any of which may have loci executing within them. At any time, a locus can only address the data visible in the container in which it is executing. Each container is independent and not part of some

larger structure, allowing different management techniques to be implemented for each container.

Facilities must be provided which allow the transfer of data between containers. The mechanisms provided in Grasshopper are mapping and invocation. These are described in the following sections.

7.4.2. Capabilities and Protection

The protection abstraction provided by Grasshopper is the *capability* [Fabry 1973]. The capability provides both a referencing mechanism for entities within the system, and protection of access to these entities. An operation can only be performed by the presentation of a valid capability which references the entity upon which the operation is to be performed and which permits the requested operation. An entity can be identified by many capabilities, each individual capability may endow its owner with different abilities to apply operations to the referenced entity.

Both loci and containers can own capabilities. Such capabilities are maintained in a segregated space by the kernel, thus preventing capabilities from being forged or altered. Capabilities can only be given to loci or containers through the action of the kernel. In a manner similar to that implemented by the Monads system [Rosenberg and Keedy 1987] the kernel enforces a strict protection regime in which capabilities can only be passed on to other entities with the permission of the existing holder of the capability. Thus a structured and safe referencing environment is built.

For the purposes of the remainder of this chapter, and in particular, discussion of specific procedure specifications, it is sufficient to regard capabilities as kernel provided and protected references. User level code is able to indirectly refer to those capabilities it has access to by using a *Capref*. A Capref is simply an index into the kernel maintained list of capabilities owned by an entity. Although Caprefs can be arbitrarily created or altered they can only make use of those capabilities actually installed within an entity, thus preserving safety in the system.

7.4.3. Loci

In Grasshopper, loci are the abstraction over execution (processes). In its simplest form, a locus is simply the contents of the registers of the machine on which it is executing. Loci are maintained by the Grasshopper kernel and are inherently persistent.

Throughout its life, a locus may execute in many different containers. At any instant in time, a locus executes within a distinguished container, its *host container*. The locus perceives the host container's contents within its own address space. Virtual addresses generated by the locus map directly onto addresses within the host container. A container comprising program code, mutable data and a locus forms a basic running program. Loci are an orthogonal abstraction to containers. Any number of loci may execute within a given container; this allows Grasshopper to support multi-threaded programming paradigms.

Any container may include as one of its attributes a single entry point known as an *invocation point*. When a locus invokes a container, it begins executing code at the invocation point. The single invocation point is important for security; it is the invoked container that controls the execution of the invoking locus by providing the code that will be executed.

7.4.4. Container mappings

The purpose of container mapping is to allow data to be shared between containers. This is achieved by allowing data in a region of one container to appear in another container. In its simplest form, this mechanism provides shared memory and shared libraries similar to that provided by conventional operating systems. However, conventional operating systems restrict the mapping of memory to a single level. Both VMS [Levy and Lipman 1982] and variants of Unix provide the ability to share memory segments between process address spaces, and a separate ability to map from disk storage into a process address space. The Mach and Chorus operating systems [Acceta, Baron et al. 1986; Abrossimov, Rozier et al. 1989] provide the notion of a *memory object*, which provides an abstraction of data. In these systems, memory objects can be mapped into a process address space, however memory objects and processes are separate abstractions. It is therefore impossible to directly address a memory object, or to compose a memory object from other memory objects.

By contrast, the single abstraction over data provided by Grasshopper may be arbitrarily recursively composed. Since any container can have another mapped onto it, it is possible to construct a hierarchy of container mappings as shown in Figure 1. The hierarchy of container mappings form a directed acyclic graph maintained by the kernel. The restriction that mappings cannot contain circular dependencies is imposed to ensure that one container is always ultimately responsible for the data. In Figure 75, container *C2* is

mapped into container *C1* at location *a1*. In turn, *C2* has regions of containers *C3* and *C4* mapped into it. The data from *C3* is visible in *C1* at address *a3*, which is equal to *a1* + *a2*.

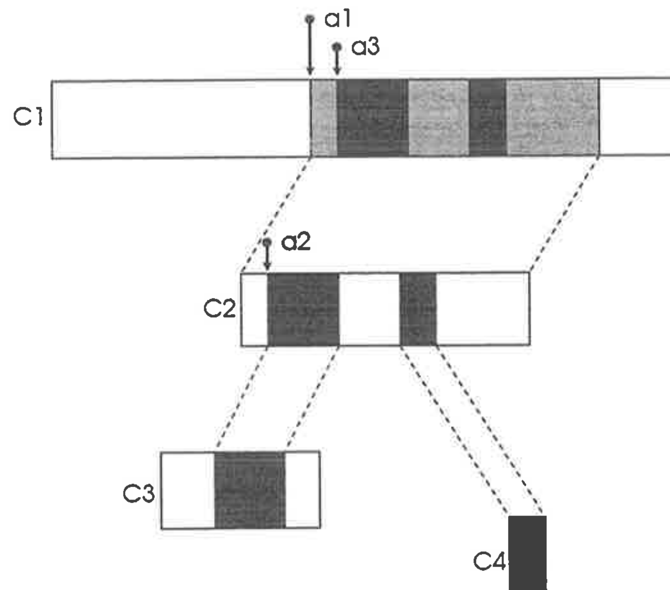


Figure 75: A container mapping hierarchy

7.4.5. Managers

Thus far we have described how all data storage in Grasshopper is provided by containers. However, we have not described how data becomes visible within a container nor how persistence of this data is maintained. When data in a container is first accessed, the kernel must provide the concrete data that the container represents. Managers are responsible for providing the required data to the kernel and are also responsible for maintaining the data when it is not resident in addressable memory. In Grasshopper, the manager is the only mechanism by which data migrates from stable to volatile storage. Rather than being part of the kernel, managers are user level programs which reside and execute within their own containers; their state is therefore also resilient. Managers themelves have their state managed by another manager. The fixed point in this recursive managment is provided by managers that are able to manage their own data. The concept of a manager is similar to the Mach external pager. In common with Mach and more recent systems [Harty and Cheriton 1992; Khalidi and Nelson 1993], managers are responsible for:

- provision of the pages of data stored in the container,
- responding to access faults, and
- receiving data removed from physical memory by the kernel.

In addition, Grasshopper managers have the following responsibilities:

- provision of stability for the container data, and
- maintenance of coherence in the case of distributed access to the container

When an executing locus attempts to access data at a virtual address for which no physical page is mapped the kernel translates the resultant page fault into an invocation of the appropriate manger. Making data accessible in a container takes place in two steps:

i. the manager associated with a particular address range is identified, and,

ii. the appropriate manager is requested to supply the data.

The kernel is responsible for identifying which manager should be requested to supply data. This is achieved by traversing the container mapping hierarchy. Once the correct manager has been identified, the kernel requests this manager to supply the data. The manager must deliver the requested data to the kernel, which then arranges the hardware translation tables in such a way that the data is visible at an appropriate address in the container.

Managers are responsible for maintaining a resilient copy of the data in a container on stable media. It is only within a manager that the distinction between persistent and ephemeral data is apparent. Managers can provide resilient persistent storage using whatever mechanism is appropriate to the type of data contained in the managed container. Since managers are responsible for the storage of data on both stable media and in addressable memory they are free to store that data in any way they see fit.

7.5. Resilience

Grasshopper needs to provide an intrinsic mechanism by which the system can always recover a resilient state. In line with the goals of orthogonal persistence, this must be achieved without requiring user code to participate in the mechanism, or indeed allowing user code to perceive the mechanism at all. Users of containers should only perceive a resilient persistent address space. Furthermore, Grasshopper seeks to provide this mechanism in such a way that individual container managers may use their own optimised stability protocols and still co-exist within the Grasshopper framework. In particular the mechanism provides support for managers that:

- provide simple snapshotting mechanisms,
- provide incremental snapshot generation,
- utilise difference logs, and

- utilise high level replay logging,

to maintain a resilient persistent state of the containers in their charge. It is also important that the mechanism allows each manager maximum freedom for asynchronous action, avoiding regimes where large numbers of managers are required to simultaneously meld the volatile data in their charge. Support of logging aids this goal since logging allows the manager greater freedom in the generation of stable states. It also allows the manager to provide a large range of stable states , thus as described in chapter 3, the operating system is afforded greater freedom in identifying and creating consistent global states.

Creating a design in which such freedoms are provided requires that the system is able to accurately track and characterise the nature of interactions between the supported computations.

7.5.1. Causality

A central problem is the impossibility of providing universal time in a distributed system. The Casper system described in Chapter 6 is able to track interdependencies easily because all communication is serialised through a single central server. In a system built from many separate stores and independent computations, this is no longer feasible. Instead, the notion of *causality* is useful for reasoning about distributed computations. Causality can intuitively be defined in terms of some entity being in some way dependant upon the state of another. For this to happen, state information must be passed between entities.

More formally, causality may be captured using Lamport's *happened-before* relation [Lamport 1978]. Briefly, the happened before relation "$\rightarrow$" is the smallest relation such that:

If a and b are events in the same entity and a comes before b then $a \rightarrow b$

If a is the sending of information from one entity and b is its receipt in another then $a \rightarrow b$

If $a \rightarrow b$ and $b \rightarrow c$ then $a \rightarrow c$

If $a \not\rightarrow b$ and $b \not\rightarrow a$ then a and b are said to be concurrent.

Causality in distributed systems is very similar to relativistic effects in the physical world caused by the finite speed of light. It is impossible for some action occurring at a distance to affect an observer until results of the action traverse the space between the two. Often the results of some event are represented as a *light cone* radiating out from the event [Hawking 1988]. Similarly the domain of events that can affect a point may be presented

by a *causal cone* which expands backwards in time. In Figure 76 only those actions that occur within the cone can affect the point represented at the apex of the cone. Action α occurs within the causal cone and its effect is visible at χ. However β lies outside the causal cone and therefore χ cannot be causally dependant upon it. In a similar manner the causal cone in a computational system may be modelled.

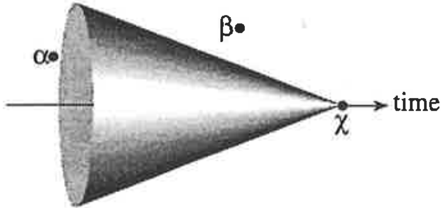


Figure 76. A causal cone in space time. α can affect χ but not β.

Figure 77 represents a distributed computation in which each horizontal line represents a single node with time increasing to the right. Communication between entities is represented by a directed arc between time lines. In Figure 77, entity N3 at time χ is affected by an action α in N1, but action β is invisible to it. Thus χ is causally dependant upon α but not β.

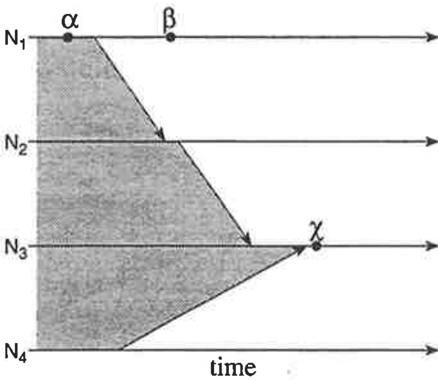


Figure 77. A causal cone in a distributed system. Event α can affect χ but not β.

An operating system that provides orthogonal persistence must provide some failure recovery mechanism. In this discussion, as in the rest of this thesis, we assume that the system is fail stop. That is, if a failure occurs, that component will cease to operate. Upon recovery, the system must find some consistent system state from the separately checkpointed states of system components. The problem is how to generate and find a set of these committed states that form a consistent global state; such a subset is known as a *consistent cut*.

Upon recovery, a useful consistent cut is one that represents some possible correct system state. This need not represent the system as it actually existed at some moment in

time, indeed it probably does not. It must, however, represent a possible state; that is one reachable through some correct execution of the system components.

Formally a consistent cut is a subset of the events which comprise the system such that for all events in the cut, if e is an element of the consistent cut and e' $\rightarrow$ e then e' is also an element of the cut [Mattern 1990].

Graphically, consistent cuts are easily conveyed as a line drawn downwards through a time diagram; dividing the diagram into two parts with the past on the left and the future on the right. A cut is consistent if no arrow (a communication) starts in the future and ends in the past, and thus no message is received before it is sent. The intersection of the cut line and a node's time line represents the time at which a snapshot of the state of the node was committed.

In order to maintain consistency, either outgoing messages must be recorded as part of the state of a node or a cut may not cross any message arrow. In either case, a message is never lost. Figure 78 shows four possible cuts in which α and γ are consistent cuts and δ is not. If messages in transit are recorded, β is also a consistent cut otherwise it is not.

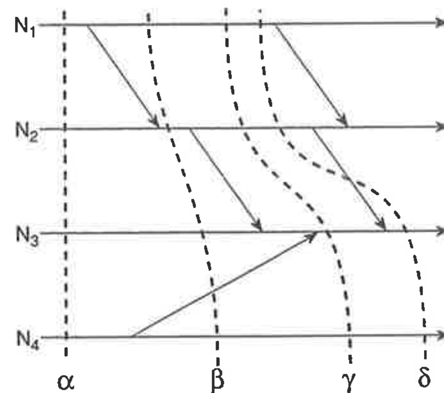


Figure 78. Consistent and inconsistent cuts.

7.5.2. Lamport Time

Lamport [Lamport 1978] describes a method of providing a global clock by which causal relationships can be characterised. Each event is tagged with an integer and the *happened before* relationship is implemented by comparing the tags for different events. As shown in Figure 79, using Lamport time, each entity maintains a counter, initially zero. At each atomic action, which includes the transmission and reception of messages, the counter is incremented. All messages are tagged with the counter's value. Upon receipt of a message, the local counter is set to the maxima of the local counter and the counter in the message.

Using Lamport time, events with the same time in different nodes are arbitrarily ordered using some secondary criteria such as entity identification.

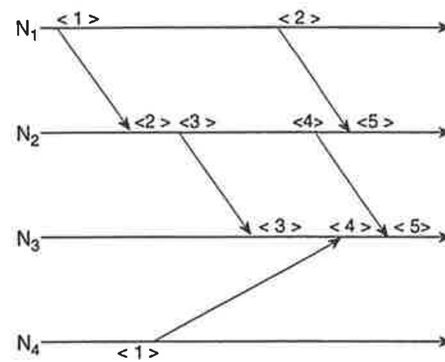


Figure 79. Lamport Time.

In a distributed system, message passing occurs concurrently in separate parts of the system; therefore the ordering of events is characterised by a partial ordering. However, Lamport time presents a single total ordering of events which is one of many possible correct total orderings. Extending Lamport time to express the partial ordering inherent in a distributed system leads to *vector time*.

7.5.3. Vector Time

Vector time has been described by Fidge [Fidge 1988] and Mattern [Mattern 1989] and may be viewed as an extension of Lamport Time. Rather than keeping a single counter, each entity within the system maintains a time vector, in which each element represents knowledge about other entities within the system.

Vector time is maintained as follows; each vector has as many elements as there are entities within the system. $VT_i[j]$ denotes the jth element of the time vector for entity i.

Each element of every vector is initially zero.

On each atomic action (including message receipt and transmission), the entity increments the element of its vector corresponding to itself, that is $VT_i[i] = VT_i[i] + 1$.

Whenever a message is sent to another entity, the sender's vector is transmitted with the message.

Upon receipt of a message, the receiving entity updates its own vector as follows. If any element of its own vector is greater than the corresponding element of the vector received, it is untouched. If an element of the received vector is greater than the

corresponding element of the receiver's vector, the receiver updates that element to equal the received value.

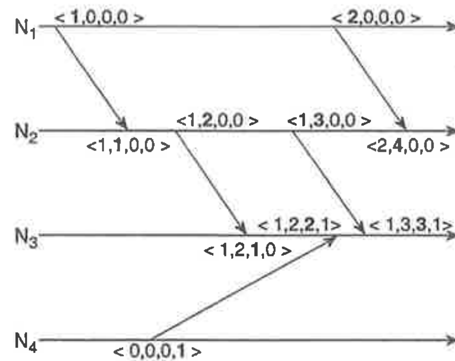


Figure 80. Vector Time.

Figure 80 shows a simple example of vector time. Using vector time, event α in entity N_i happens before event β in entity N_j iff,

$$VT\alpha[i] < VT\beta[i]$$

In effect this says that entity i must have communicated with entity j, perhaps via some intermediaries. In addition to the happened before relation, vector times also encompass the notion of concurrency. Two events α and β are concurrent if

$$\alpha \not\rightarrow \beta \text{ and } \beta \not\rightarrow \alpha$$

For example, the nodes labelled <1,0,0,0> and <0,0,0,1> are concurrent events. Unlike Lamport time, vector time captures the partial ordering of events within a system. Thus, vector time captures the *happens before* relationship and the notion of concurrency in the system. It therefore contains all the information needed to formulate a consistent cut.

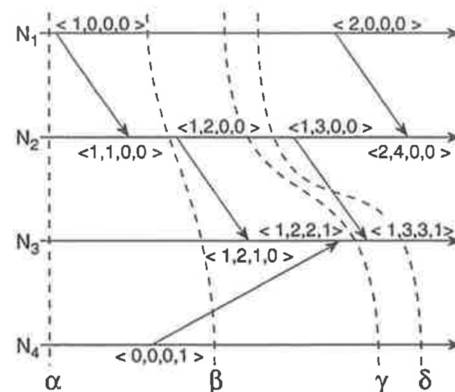


Figure 81. Consistent cuts and vector time

Vector time may be used to characterise consistent cuts. Johnson and Zwaenepoel [Johnson and Zwaenepoel 1990] show a method which involves the construction of a *dependency*

matrix which is a matrix whose rows correspond to the vector times of all nodes in a system. For example, the matrix for the cut δ in Figure 81 above is:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & \mathbf{2} & 0 & 0 \\ 1 & \mathbf{3} & 3 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

They show that a dependency matrix M represents a consistent cut iff,

$\forall i,j \; Mij \leq Mjj$.

In other words, the elements in the column j must be less than or equal to the diagonal element. This says that no entity i depends upon an event that has happened after the consistent cut was established. The cut denoted by δ is not consistent since entity three's value for entity two (M[2,3] = 3) is greater than entity two's own value (M[2,2] =2). This represents the reception of a message by entity three without the corresponding message send taking place.

7.5.4. Vector time in Grasshopper

Conceptually the Grasshopper kernel maintains a time vector for every container and every locus in the system. This vector contains as many entries as there are entities within the system. In reality, within the kernel, vectors are represented by entity-id, time-stamp pairs. Initially, each entity's vector contains a pair representing itself. Additional pairs are added whenever entities interact.

The kernel implementation of time stamps is hidden from user level programs for two reasons:

- Entity names are an implementation dependant construct, there is no reason for their representation to be visible outside of the kernel.
- If user level code (such as in managers) has direct access to time stamps, malicious code could alter or forge them. This could result in user level code forcing the rollback of parts of the system over which they do not have authority.

The kernel therefore provides access to time stamps via capabilities which act as a proxy for the time stamp.

An optimisation may be made by observing that whilst executing, a locus is so intimately bound with its host container, that it makes no sense to distinguish between the time stamp of the locus and that of the host container. Therefore whilst executing, a separate vector for a locus need not be maintained. When the locus is not executing it

ceases to be causally tied to the container. However, the maintenance of vectors during scheduling and de-scheduling of loci is very expensive. We therefore consider the bond between locus and host container to continue even when the locus is blocked. Once this step is taken there is no need to maintain a time stamp for a locus. The only proviso is that a locus must always have a host container (thus ensuring that it always has a time vector). Invocation is therefore an atomic operation. This atomicity is similar to that used by Clouds [Dasgupta, LeBlanc et al. 1988]. Since invocation is the only communication mechanism provided by Grasshopper, once invocation becomes atomic, it no longer makes sense for a consistent cut to cross a message send. This further simplifies the notion of a consistent cut.

A container's vector time is updated whenever it is invoked by a locus. When a locus leaves a container it takes with it a copy of that container's vector time. The kernel updates the invoked container's vector clock with this time vector using the normal vector time update mechanisms described above.

7.5.4.1. Mapping

Containers may also become causally interdependent due to the results of mapping. The act of mapping does not itself cause interdependence; interdependencies occur when mapped regions are accessed by an executing locus. A locus becomes dependant upon the state of a container by reading data from it, a container becomes dependant upon a locus through being written upon by the locus. Thus containers become interdependent through the execution of loci which have access to them. A range of techniques may be used to update the vector clocks to reflect such interdependencies; these vary from simplistic coarse grained to complex fine grained mechanisms. Some possible tactics include:

- *Eager merging time vectors upon mapping.* As soon as a mapping is established the kernel can elect to merge the vector times of those containers which are now visible within the same address space. Merging consists of creating a new time vector by the update of one containers time vector by the other, and then making both containers refer to this vector for all subsequent use. Eager mapping is a very pessimistic approach, but one which requires a minimum of kernel effort.
- *Lazy merging of vectors on map.* An optimisation of the above, the kernel waits until the first access to data in the mapped container before merging the vectors. This requires that the kernel's virtual memory system is able to detect such

accesses. This can be accomplished by either using the access protection mechanisms of the virtual memory, or by detecting loading of translation lookaside buffer (TLB) entries in those architectures which feature software control of TLBs.

- *Lazy merging with asynchronous unlinking*. This is a further addition to the previous tactic. Rather then leaving mapped containers with merged time vectors, the system occasionally splits the time vectors, and resets the mechanism used to detect access to a mapped container. Splitting creates two separate time vectors each referred to by one of the separated containers. The more often splitting is performed the finer grained and more accurate the representation of causal interdependencies will become.

Further tactics exist for tracking interdependencies: at the most extreme the system can maintain vectors, and track causal dependencies for each individual page of data in the system. These tactics are the subject of vigorous ongoing research [Jallili and Henskens 1995] and beyond the scope of this discussion.

7.5.5. Stabilisation

The key to resilient stable storage in Grasshopper is the interaction between the kernel and the container managers. Managers are responsible for creating recoverable copies of the contents of a container. It is the kernel that is responsible for the co-ordination of these stable states, creating and maintaining a recoverable system state. In a distributed system the separate kernel instances co-operate to maintain this recoverable state.

To this end the kernels must maintain in their own resilient stores enough information to allow each container manager to recover the internal state of the containers in their charge. This must be done in such a way so as to ensure that after a system failure a self consistent system state can be rebuilt. A recovered system state need not be the same as any state that actually existed before a system failure, but it must be one of the possible system states, as might have been generated by some correct execution of the system.

7.6. Maintaining Global Consistency

Many different methods for maintaining a globally consistent state for persistent systems exist. These designs are guided by two orthogonal issues: the cost of failure and whether deterministic execution is possible.

Methods that take account of the cost of failure can be characterised as optimistic or pessimistic. Optimistic algorithms assume that failures occur with sufficient rarity, that it is better to place the bulk of the work needed for system recovery in the recovery phase, speeding up normal execution. Pessimistic systems place the main burden of work needed to maintain a recoverable state into the normal execution of the system. This may be because they take the view that failures are quite likely, or that the need to recover quickly from failures is important. Systems which require the user view to be one of total reliability also place a greater burden on normal execution and can also be characterised as pessimistic.

Determinism of execution is important when systems recover through replay. In a system in which all the components execute in a deterministic manner almost arbitrary recovery may be performed. In principle, in a deterministic system, a program can be restarted from an arbitrary point and it will recover to the point at which the system failed.

Even if a deterministic program interacts with non-deterministic external agents, replay can still be used as a recovery mechanism. This may be achieved by logging all incoming messages to stable media and keeping a stable counter of outgoing messages. Upon system restart, the logged incoming messages are replayed and output is discarded until the number of output messages equals the stable counter.

The manager of a container in Grasshopper must therefore be able to provide the following functionality.

- The ability to create a snapshot, recoverable from stable media, of the managed container. This is the minimum required functionality.
- If the container is managed using replay logs the manager should be able to inform the operating system which additional states it is able to recreate. This allows the global consistency management algorithm greater freedom in determining a global consistent cut.

However, if a program does not execute in a deterministic manner, it is not possible to use replay to recover from failure. The only recovery mechanism that may be employed is

checkpointing. Tactics for providing deterministic execution in a manner as transparent to the user as possible are surveyed in Section 7.8.

7.7. Stability and Resilience

Managers are responsible for the persistence of the data which they manage. The manner in which this is achieved is at the discretion of the implementor of the manager. As described above, the existence of a recoverable globally consistent state is the responsibility of the Grasshopper kernel. To implement such a regime, the manager to kernel interfaces provide functions to explicitly support recoverability. The design of this interface has aimed to provide the managers with considerable freedom in the manner in which they maintain stable state and in the manner in which they recreate stable state upon demand.

Initially discussion of the interface will be done in terms of coordinating simple snapshots, we will then describe the manner in which logging and replay may be integrated.

7.7.1. Initiation

A manager may at any time elect to take snapshot of the data in its charge, however for this data to be useful it must cooperate with the kernel to ensure that the data is both good and useful. For data to be good the kernel must ensure that no modifications to the container data occur during the time in which the manager creates the snapshot. To be useful, the data must be able to form part of a global consistent cut. This cut need not exist immediately, the system may implement algorithms which lazily create new consistent cuts.

The stabilisation sequence is initiated by any locus making a request of the kernel. It is this request to the kernel that both allows the manager to be sure that the data it generates is consistent and allows the kernel to track the times at which the managers have taken snapshots. The request to the kernel takes the form of a *snapshot_request* call which has the following signature:

snapshot_request(**Capref** cont_id)

Often the locus making the *snapshot_request* call will be a locus executing within the manager of the container, however this need not be so. External agents may also need to initiate the creation of snapshots. When *snapshot_request* returns to the requesting locus a snapshot will have been created. The kernel may also elect of its own volition to initiate

snapshot creation, this may be in response to pressure on physical resources or the result of a particular coordination strategy used by the kernel.

To effect the snapshot sequence the kernel first ensures that no loci may modify the container until the snapshot is complete. This is achieved by ensuring that no loci are scheduled in the container being snapshotted. Next the kernel instructs the manager of the container to take a snapshot of the container.

7.7.2. Snapshotting State

The kernel requests a manager to take a snapshot of a container by making a call on the manager's *container_snapshot* interface which has the following signature:

container_snapshot(**Token** cont_id ; **StateInterval** state_interval)

The container to be stabilised is specified by *cont_id*. This is a previously agreed upon token by which the kernel and manager recognise requests the specific container for which a request is being made. Since many snapshots may exist, an unambiguous way of identifying them is needed. This is provided by the kernel through the parameter *state_interval*. The value of state_interval is essentially the containers own ordinal value from the time vector maintained by the kernel for the container. When this call is made the manager must create a snapshot of the container. However, this snapshot need not be stable and may be implemented in any fashion the manager chooses. For example, the manager may choose to protect from write access the entire address range and lazily make either volatile or stable copies of individual pages as access violations occur. Furthermore, much of the data required to form the snapshot may already be available within the manager. A more eager approach is for the manager to make a volatile copy of all the container data in main memory or a stable copy on disk.

When the call returns, the kernel takes a snapshot of any kernel level data structures associated with the container. Of particular importance are the loci that are *running* in the snapshotted container. The kernel makes a snapshot of these by taking a copy of the saved register sets and other kernel data structures associated with the loci. Upon recovery from failure, those loci state snapshots associated with the container snapshot used for recovery provide the state needed to reconstruct the loci. The provision of kernel stability is discussed in detail in [Lindström, di Bona et al. 1994]

On return from the *container_snapshot* call, the kernel is assured that at least a volatile snapshot has been created and the kernel is again free to schedule loci to execute in the container. Use of volatile snapshots can increase parallelism and asynchrony in the system

by providing the kernel with a greater choice of snapshots from which to construct consistent states without the extra cost incurred in creating stable snapshots.

7.7.3. Co-ordination

To co-ordinate globally consistent stability the kernel needs knowledge of which container states are stable and the ability to request that some currently volatile snapshot be made stable. The following calls provide this functionality.

Once a snapshot is made stable, the manager may convey this information to the kernel using the *state_interval_stable* system call. This call has the following signature:

state_interval_stable(**Capref** cont_id ; **StateInterval** state_interval)

However, managers are not obliged to make the *state_interval_stable* call when a state interval has been made stable. To allow the kernel to discover the state of any state interval for itself the call *state_interval_query* is provided. It has the following signature:

state_interval_query(**Token** cont_id ; **StateInterval** state_interval
StateIntervalState *interval_state)

The parameter *interval_state* allows the manager to return an enumerated type (StateIntervalState) conveying whether the state is volatile, stable or non-existent. Like the other calls, a container identifier and a state interval index is used to identify some point of time in the history of the container.

When a manager receives this call it is not obliged to make the interval stable; it may however be a good policy to do so since the kernel is likely to be trying to form a consistent cut incorporating this snapshot. Generating a stable copy of a volatile snapshot may aid the generation of a new recoverable global system state.

The kernel may also force managers to make a snapshot stable. This is provided to the kernel through the *container_stabilise_request* call which has the following signature:

container_stabilise_request(**Token** cont_id ; **StateInterval** state_interval)

Following this call and before returning to the kernel, the manager is *required* to ensure that the state of the container at the time denoted by *state_interval* is recorded on stable media. However, it is likely that not all snapshots taken by a manager will be useful or will exist indefinitely. A manager may choose to discard a snapshot to free space to enable a new snapshot to be generated. Indeed this may be unavoidable in store designs which implement a bi-phase policy, where at most two stable states exist for a store. The kernel must be informed of the destruction of a snapshot in advance, preventing the kernel from

using a stable snapshot that no longer exists as part of a consistent cut. To support this the kernel provides the following call:

snapshot_discard(**Capref** cont_id ; **StateInterval** state_interval)

7.7.4. Progress

As the system executes the kernel will attempt to create new consistent recoverable versions of the system from the stable states made available by container managers. When the kernel finds such a globally consistent state it must inform the managers of the state intervals and containers that form part of that state. This is provided by the *snapshot_is_recoverable* function provided by every manager:

snapshot_is_recoverable(**Token** cont_id ; **StateInterval** state_interval)

When a manager receives this call it is free to discard all volatile, stable and recoverable snapshots that occur before the time denoted by *state_interval* for the container denoted by *cont_id*. The manager may not, under any circumstances, discard the state of the container denoted by the state interval until it receives another *snapshot_is_recoverable* call with a higher state interval. It is with this function that the global state of the persistent system makes progress. The states of containers and their progress from a volatile to stable to recoverable state is shown in figure 82.

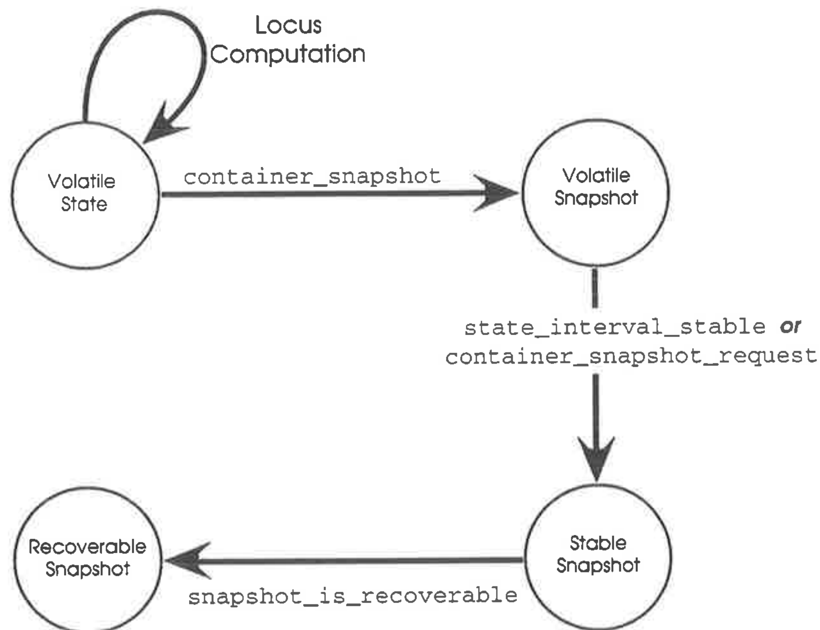


Figure 82. Stable state transitions

7.7.5. Failure Recovery

In a single node version of Grasshopper, failure recovery following a node crash is relatively simple. The system will reboot and restore the state of the loci which were in existence at the time of the last consistent cut. These loci will fault against pages visible from their host container and the page faults are delivered to the appropriate managers. However since managers may have stored several different versions of the state of a single container on stable media, they require guidance regarding from which stable checkpoint to fetch the pages. Managers provide the interface function:

cont_rollback(**Token** cont_id; **StateInterval** state_interval)

This function is called by the kernel for each container following a reboot to instruct each manager of the correct state interval of the container. This call also allows the manager to tidy up stable data structures. When a manager receives this call it is assured that none of the stable states stored on stable media, other than the state denoted by *state_interval* serve any purpose. The checkpoint denoted by *state_interval* forms part of the most advanced consistent cut therefore earlier states are of little use. Similarly, checkpoints in advance of *state_interval* are in the *previous* future and are also of little use. If a useful state existed with a *state interval* greater than state_interval the kernel would have used it.

7.8. Logging, Determinism and Proxies

We have seen that the kernel can find consistent cuts by searching through the periodic container snapshots made by managers. In practice, if inter-communication is common it may be hard or impossible for the kernel to lazily find a consistent cut. The kernel can of course, force consistency by use of the *container_stabilise_request* call. However, under some circumstances, with more information, the kernel is afforded greater freedom in finding globally consistent states.

Snapshots represent the state of a container at one moment in time. This makes their use limited in the formation of a consistent cut. An otherwise useful consistent cut may have existed just prior to a snapshot, or just after a snapshot. If a snapshot of a container has been taken and the execution pattern within the container is deterministic, the state of the container at any time after the snapshot is reconstructable from that snapshot using replay. The kernel makes use of knowledge of such behaviour, giving it greater freedom in the construction of consistent cuts.

The interfaces described earlier are explicitly designed to support such algorithms. For instance, a manager may log all invocations entering a container. Each time an invocation occurs the container starts a new period of deterministic execution dependent upon the reception of the invocation. This period is termed a *state-interval* following the terminology of Strom and Yemini [Strom and Yemini 1985]. These are the state-intervals used to tag container states in the above interfaces and are maintained by the kernel. Each time a state-interval is generated it is a potential candidate for inclusion within a consistent cut, so long as the manager can guarantee the kernel that the data extant at the generation of the state-interval can be re-generated on demand after a system failure. The calls *state_interval_stable* and *state_interval_query* allow the managers to make such guarantees. The logging manager in the above example need only keep an initial stable snapshot and periodically make stable copies of its log. It is able to assure the kernel that the container state is stable up to the state-interval of the last entry in the stable log.

Managers inform the kernel of their respective abilities by use of a system call which has the following form:

service_mechanism(**Capref** cont_id; **Service** service)

Using this call, a manager may inform the kernel if it is snapshotting, transactional, or logging. This informs the kernel that the container *cont_id*, can be recreated at times other than just those represented by snapshots.

7.8.1. Deterministic Execution

Replay of events to recreate a containers state requires that execution within the container must be deterministic. In Grasshopper non-deterministic behaviour can occur in one of four ways:

1. Concurrent execution of more than one locus at a time within a container.
2. A locus may make a system call or invoke another container and return with unreproducible data.
3. A new locus may invoke the container, bringing with it external state.
4. Concurrent sharing of modified data container through container mapping.

These activities are legitimate activities for user level programs, and should not be prohibited if at all possible. Here we discuss mechanisms which provide such activities without compromising deterministic execution and replay.

To ensure deterministic behaviour, managers need to be in control of the behaviour of the following:

- all loci entering, exiting and executing within a deterministic container.
- all mappings by which shared data may be seen within a deterministic container.

7.8.1.1. Proxies

Tracking of loci entering and leaving a container is achieved by setting a *proxy* through which all incoming and outgoing locus movements are routed. Proxies are set using the *set_proxy* system call which has the following signature:

set_proxy(**Capref** cont_id ; **Capref** proxy_id)

Once this call has been made, any invocations incoming to *cont_id* are passed to the container designated by *proxy_id*. Similarly any outgoing invocations (including system calls) from *cont_id* are sent to *proxy_id*. To enforce deterministic execution a container manager will insert itself as the container's proxy. Such a manager is informed of events via the *proxy_invoke* call which has the following signature:

proxy_invoke(**Token** cont_id ; **StateInterval** state_interval ; **Bool** direction ;
invoke_param1, invoke_param2....)

This call specifies the proxied container in which the event occurred, the state interval at which the event occurred, whether the event was incoming or outgoing, and the parameters to the original call.

Thus managers are notified of a locus' intent to leave or enter a deterministic container. In the case of outgoing invocations, the manager may choose to make a note of the fact that the invocation has happened and the appropriate state interval. It can proceed with the invocation on behalf of the managed container and record the results of the request. On replay following a failure it can ensure that this event is sequenced at the same time relative to other events in the history of the container, and if need be interpose the results of the original request, ensuring that replay of invocations from within the managed container always replay the same history.

7.8.1.2. Concurrency

Use of the proxy mechanism as described above allows a manager to ensure that all invocations both incoming and outgoing occur in the same order and provide the same data on replay. To provide the illusion of concurrent execution of a number of loci within a

container, whilst maintaining determinism, is also possible through use of the proxy mechanism. This is achieved in a number of ways.

- The perception of concurrent execution within a container by a number of loci can be simulated if each locus is allowed to execute within the container for a short period and then replaced by another locus, so long as the scheduling of these loci is deterministic. Such a scheme may be implemented with the proxy mechanism.

  The container manager acts as the proxy for the container (something which it must already do to effect the replay of invocations described above.) However it retains each locus in a stalled state within itself. The manager allows these loci to run in the managed container one at a time. Since the manager will also receive all outgoing invocations it can use these requests to provide a deterministic time at which to remove one locus and allow another to continue execution. To guard against starvation (if the running locus does not perform any invoke calls for some time) the user code in the managed container should occasionally invoke a special *yield* invoke call which allows the manager to deschedule the executing locus. The need for this call does slightly compromise the transparency of the mechanism.

- The second approach is for the manager to provide separate execution spaces for each locus through surrogate containers. This approach is similar to that described by Wu and Fuchs [Wu and Fuchs 1990] for hardware supported reliable shared memory. Each locus perceives the contents of the invoked container using a read only mapping. If this approach is employed read/write conflicts which would violate determinism may be detected by trapping access exceptions. When violations occur, the contents of all surrogate containers can be snapshotted to form a snapshot of the base container and execution allowed to proceed once more. Although apparently expensive this mechanism is no costlier than many transaction models proposed for persistent systems, and since only pages modified since the last such commit need be copied the expense need not be great.

7.8.1.3. Mapping

The final restriction (no sharing of modified data) may be provided in one of two ways:

- The manger can reject mapping requests which would allow modifiable data to be shared with a deterministic container. This is an unfortunate tactic since it allows

the implementation mechanism for persistence (deterministic replay) to become visible to, and change the programming model of user code. It therefore arguably compromises orthogonal persistence.

- A regime similar to that described above utilising the Wu and Fuchs mechanism cited above can be provided by the managers of any shared data. This is a greater imposition on the managers since the manager of a container mapped into view within the deterministic container must cooperate with the manager of the deterministic container. This cooperation requires each manager to make a snapshot of any container from which shared pages are mapped upon any attempt to modify the data on that page.

7.9. Distribution

As described earlier, the Grasshopper system seeks to provide an environment which abstracts over locality. For example, a locus may invoke, or map, any container for which it holds a capability; regardless of the node on which the locus is executing. This has the consequence that loci executing on many nodes may transparently and concurrently access the data of a container.

The user model presents the illusion of a single container that is available wherever it is required. In practice, this illusion is implemented by maintaining a representation of the container on each node. Earlier we stated that each container is maintained by a single manager. In reality, each local representation may refer to a separate manager. These managers collude to present the illusion of a single container and may be thought of as a single distributed manager. This coherent view is achieved by the managers exchanging pages with each other as necessary.

Each local representation of a container has associated with it its own time vector. Whenever a page is passed from one manager to another, a time vector is carried with it and the vector clock of the recipient local container representation is updated. Thus, causal dependencies arising through distributed shared memory require no further infrastructure.

A local container representation may be checkpointed independently of other representations. These checkpoints may only be used if the kernel(s) can find a consistent cut through the individual checkpoints upon recovery.

Alternatively the managers may cooperate to synchronously create a single coherent checkpoint; in doing so they will unavoidably produce a single vector time for all local representations.

7.10. Putting it Together

A Grasshopper system is capable of hosting a diverse range of programming languages and styles, and a diverse range of resilient persistent store implementation techniques. These systems are coordinated into a global whole which provides an integrated model of system resilience and reliability.

7.10.1. Simple programs

In Grasshopper even the most primitive language, for example C, may be provided with a resilient, persistent execution environment without modification to the compiler or run time system. This may be achieved in a variety of ways with varying amounts of sophistication. However, the simplest is to execute each program in a Grasshopper container with its own manager as shown in Figure 83 below.

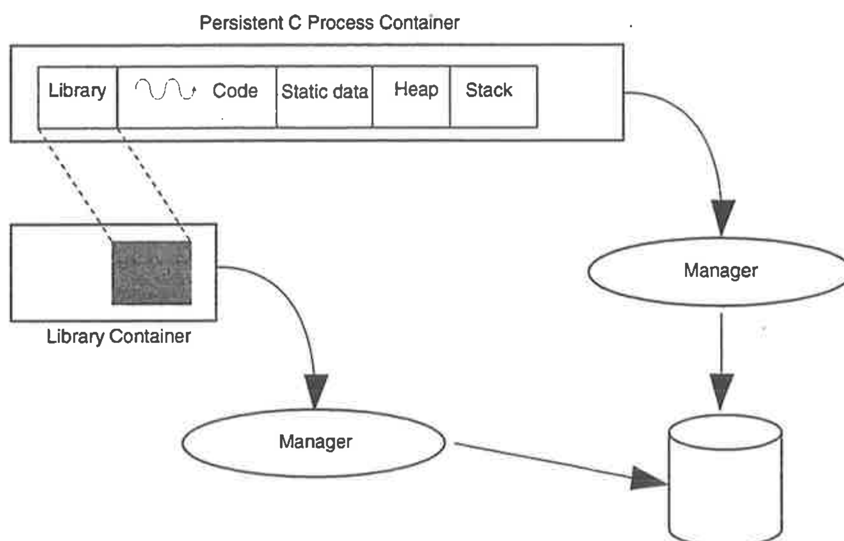


Figure 83. Running a C program under Grasshopper.

In this scheme, a C program is organised in memory in the same manner as under Unix, with code followed by static data, heap and stack space. The managers provide both the functionality of conventional demand paged virtual memory plus resilience. A manager might use a store design such as the Casper system described in Chapter 5. The manager can save the entire state of the running process on disk and restart it at any arbitrary point in its execution in a manner that is totally transparent to the running process. Libraries may be provided by mapping library code from other containers into the address space of the

process. Libraries provide both the usual Unix style libraries and code for communicating with other processes and for binding to persistent data.

7.10.2. Persistent Languages

Implementation of persistent languages may be effected in the same manner as above, simply by embedding the persistent environment inside the address of a container. However the manager is capable of aiding implementation further. The tactics discussed in Chapter 3 for expanding the apparent size of the persistent address space by pointer swizzling at page fault may be implemented within the manager of a container.

Grasshopper allows user code to protect the pages within containers against access, and provides a user level exception delivery system. Thus implementation of a language system may take advantage of the tactics described in Chapter 2 to realise transparent pointer quarantine and garbage collection.

The capability mechanism in Grasshopper allows individual containers to safely distribute to other containers the right to invoke. By controlling distribution of these rights it is possible to build a federated community of cooperating type-safe stores. Individual containers support separate stores, and are provided with the right to invoke one another. Communication through invocation can take the form of IPC, RPC [Birrell and Nelson 1984], remote evaluation calls [Stamos and Gifford 1990], or remote execution [Dearle, Rosenberg et al. 1991], and may involve the transfer of code and data fragments from store to store [Farkas 1994]. Type security is guaranteed by the controlled distribution of capabilities, program security through the type-safe language implementation.

Invocation between separate nodes is handled by the respective machine kernels transparently. Individual programs may communicate oblivious to the actual location of the communicants. In a similar manner shared access to data may be distributed. Individual managers can arrange to have access to containers made available throughout a network utilising DSM techniques. Since the kernels track the causal links between containers due to the transfer of data on pages managers do not need to track these links themselves. Thus DSM in Grasshopper is simplified relative to the distributed Casper implementation described in Chapter 6.

7.11. Conclusions

Grasshopper provides an operating system design in which orthogonal persistence is an intrinsic property. Programming languages of any kind are supported, from the most basic,

which may be simply embedded into a persistent address, through to rich type-safe languages which are afforded the security to establish distributed federated communities of persistent stores.

In the introduction to the chapter we presented four criteria for a persistent operating system Here we discuss the manner in which the Grasshopper design has met these criteria and, in addition, consider the manner in which distributed programming is supported within the persistent abstraction presented by Grasshopper.

i. *Persistent objects as the basic abstraction.*

Grasshopper provides abstractions over storage (the container,) execution (the locus) and protection (the capability.) Capabilities and loci are maintained by the Grasshopper kernel as persistent entities. Containers are maintained as persistent entities through the action of container managers. Any programming paradigm may be embedded within this environment and thus become intrinsically persistent.

ii. *Objects must be both stable and resilient.*

Stability in the Grasshopper is controlled by the kernel and managers are described above. Resilience is controlled by the kernel through its tracking of causal relationships between entities. Both the kernel and container managers maintain stable data in a such a manner that there is always a recoverable consistent system state available on stable media. As described earlier, simple "stop the world" bi-phase mechanisms can be employed to provide such resilience, but at the cost of system performance and useability. By placing tracking of causal relationships within the kernel and making the them an intrinsic part of the kernel operation the system is able to reduce the effort of maintaining a resilient state and to spread the effort through time yielding better useability.

iia *Manage the transition between long and short term memory transparently to the programmer.*

User level programs in Grasshopper exist with a space composed of containers, which are transparently provided with stable data through the action of managers, whilst the execution is effected by loci. The stabilisation protocol described earlier is transparent to user programs as is the provision of data to executing code by either the kernel (through the reactivation of loci) or container managers

(through the provision of data through the demand paging of data form the stable stores.) User level programs are able to request that a current state be recoverable (for example to effect external guarantees of transactions) but the actual transfer of data to stable store to effect such requests is invisible. Grasshopper provides a framework in which individual managers may, if they wish, provide persistence in different ways without the need to explicitly manage global consistency.

iii. *Processes must be integrated with the object space.*

Since loci are intrinsically persistent objects maintained by the kernel this objective is trivially met.

iv. *Provision of some protection mechanism.*

Grasshopper uses a variant on Fabry's capabilities to reference all entities in the system. These capabilities are unforgeable and maintained within protected kernel space. This a secure and flexible protection system is an intrinsic part of the Grasshopper design.

v. *Maintenance of the orthogonal persistence abstraction across distributed platforms.*

Since Grasshopper maintains consistent naming of objects across multiple machines reference to entities is transparent in a distributed environment. The intrinsic causality tracking extends from concurrent to distributed execution naturally, albeit with some performance degradation. Thus programs may act upon data oblivious to the location of the host machine upon which it is placed whilst Grasshopper kernel continues to maintain a transparent abstraction of resilient orthogonal persistence. To enable the container managers to operate in a distributed environment Grasshopper provides some additional system calls that enable managers to manipulate page sets held on remote machines.

The Grasshopper operating system delivers a design in which user level programs are embedded in an orthogonally persistent environment; all user level entities are intrinsically persistent. Indeed it is not possible for a normal user to write a program which is not inherently persistent.

Interaction with the outside (non-persistent) world is the only area where the orthogonally persistent programming paradigm ceases to be seamless. User programs may become aware of the nature of IO when IO crosses failures of the system. The programmer

is able to dictate a time in the execution of the program at which IO events generated by the program are guaranteed to have taken place, and the internal view of program progress coincident with an external users perception. Maintaining these guarantees is one of the central responsibilities of the Grasshopper kernel. Implementation paradigms were covered in Chapter 4, and may involve replay of output events for idepotent operations after failure, replay of input events, or more complex chained recovery systems. Whatever guarantees the programmer wishes to make are transparently coordinated by the Grasshopper kernel.

The other area where user level code becomes aware of ephemeral data is in the construction of containers. A design where container managers are outwith the kernel has mixed benefits. As a vehicle for research such a paradigm has value. It is easy and convenient to prototype new store architectures and memory management systems without the need become an expert on the operating system internals, changes can be made without he need to continually re-boot the host machine, and errors do not cause damage to other users. In a production environment similar gains are available. Third party vendors are able to construct specialised store systems, again without disruption or risk to the kernel. However there is evidence that external handler designs suffer a performance penalty and an initial enthusiasm for micro-kernel designs in commercial systems has waned [Welch 1991]. Also users may write container managers (through either intention or error) which do not provide for resilient storage of container data. In principle such users only compromise their own data, however if other users become causally dependant upon incorrectly managed data, failure of the resilience mechanism could flow to other parts of the system and cause uncontrolled roll-back or failure. This is the risk that any user takes when trusting in another user for the provision of a service. In this case the risk is inherent in communicating with the other user at all.

In general it is not envisaged that users will write their own container managers, rather that a small number of custom written managers for specific purposes will be provided, rather in the same manner as system libraries or third party software systems are supplied in conventional systems.

Overall the decision to provide for external container managers, or to provide a fixed management sub-system in the kernel is an engineering one and one that does not otherwise change the nature of the Grasshopper design.

The mechanisms we have described explicitly allow managers to exploit log based recovery techniques, but also allow the kernel based recovery mechanisms to exploit the extra freedom available from such techniques to create consistent recoverable system states. Such log based recovery mechanisms require that execution be deterministic. Rather than place special requirements upon the kernel, the use of a proxy mechanisms allows managers to control execution within containers, allowing the managers to enforce deterministic execution and to log external interactions with no impact on the structure of the code within containers.

At the beginning of this chapter we also listed Tanembaum's four components of an operating system, namely: memory management, file system, input output and process management. The Grasshopper system design takes these four components and ties them together into an unified whole. The container abstraction unifies memory management and the file system, and the causality tracking and recover management ties process management and I/O into the fold as well. The cental theme to this unification is causality, and the need to understand the temporal ordering of action with the system. Since the distinguishing factor of persistent systems is abstraction over data lifetime this should not be of any surprise.

Chapter 8. Conclusions

8.1. Summary

This thesis has examined implementation tactics for persistent systems and persistent object languages. The major topics covered can be summarised as the following:

- tactics for language level presentation of orthogonal persistence,
- tactics for store management, and
- tactics for distribution.

In particular we have focussed on the use of conventional page based hardware in each of these topics. The topics above overlap, and in recognising the nature of overlap it is sometimes possible to take advantage of it and create a hybrid with optimised performance.

8.2. Page Based Systems

In the presentation of a specialised environment it has always been attractive to manufacture specialised hardware. However this course seems flawed for a number of reasons.

- The designs are complex and usually attempt to make use of micro coded architectures. Current architectural practice suggests that such complex designs are doomed to poor performance and are often outperformed by conventional architectures that simply emulate them.
- The designs are highly language specific and thus limited in application
- Pragmatically there is no commercial application likely, drastically limiting their usefulness in propagating the results of research.

Clearly there is considerable merit in the provision of an orthogonally persistent system with a software architecture. Such an approach can exploit any advances in processor speed with no effort on the part of the implementor and is portable from one architecture to another with relatively little work. Considerable success has been acheived in the construction of such architectures, the PS-algol, Napier88, Galieo, and Smalltalk80 languages have all been very successfully implemented above such software architectures.

We have proposed a middle ground, one in which conventional hardware is used, with its attendant benefits, but in which explicit notice is taken of the page based virtual

memory hardware. To ignore this part of current architectures seems a considerable waste. Great effort has been expended in providing hardware which, although not always ideal can provide significant help in the implementation of persistent systems. This comes about for a number of reasons.

- Implementations of persistent system under conventional operating systems will always be involved with the virtual memory sub-system. Programs execute in a virtual address space and are subject to the action of the demand pager like any other program.
- The action of the demand paged virtual memory is very much akin to the needs of a transparent mechanism for movement of persistent data to and from stable media, there is clearly opportunity to turn its action to our advantage.
- The utilisation of page protection and exception delivery provides a mechanism by which the actions of a program running within a persistent environment can be transparently trapped. This provides a mechanism for implementing other functions required for the language implementation, especially garbage collection.

It is through a synergy of these three points that the benefits multiply. By integrating the management of virtual memory with that of movement of data to and from stable storage we can eliminate the costs associated with managing a separate swap space.

8.3. Tactics for Language Level Presentation

In approaching language level presentation the major topics covered were

- data movement,
- address generation,
- pointer representation and object formats,
- garbage collection, and
- address space expansion through swizzling.

These topics have proven to have deep interdependencies. In particular the relationship between data movement and garbage collection is especially important when persistence is a result of reachability. The value of techniques originally developed to support generation garbage collection are especially applicable.

8.3.1. Data movement

Abstracting over the location of data is one of the cornerstones of orthogonal persistence. Implementations of persistent systems using pure software realisations of a virtual machine have usually performed software residency tests, activating data movement as appropriate.

Alternatively, data movement may be entirely handled by page faulting, eliminating the software residency check, this may be done in two ways.

- Providing a persistent environment that fits within the host machine virtual address space and directly faulting pages from a page based persistent store
- Ensuring that pointers resident in addressable memory that refer to non-resident objects are redirected to "fault blocks" that are protected from access.

Systems which embed the programming environment are able to allow executing code to execute oblivious to the nature of the data movement support system. The restriction that the persistent environment fit within the host architectures address bounds can be lifted through the use of swizzling techniques (summarised below).

8.3.2. Swizzling and Addressing

Techniques in which the pointers are overwritten whilst resident in addressable memory allow the persistent address space to be arbitrarily large. Pointers with the persistent store can be larger than that supported by the host architecture whilst overwritten pointers refer to objects resident in addressable memory. This technique (termed swizzling) may be used eagerly, at the time that objects (or pages) are loaded into addressable memory or may be delayed until the pointer is dereferenced. Eager schemes can guarantee that an executing program only ever sees valid pointers. These schemes avoid any need for action by running programs to deal with unswizzled pointers. However such schemes must allocate address ranges for every object potentially addressed by in-memory objects. Further utilising page protection, techniques a hybrid system can be built in which swizzling can be delayed, avoiding such greedy allocation of address space and still avoid any interaction with user code.

8.3.3. Pointer formats

When the support systems for data movement and garbage collection operate they must be able to find the points within the data being manipulated. Most persistent languages use a self describing object format which allows pointers to be found when an object is presented. When manipulation is performed at the page level a further mechanism is

needed. Crossing maps used by some garbage collection schemes can provide this but often need to use the data on preceeding pages. When used in a persistent system this data will often require fetching from the store. A technique that encodes the structure of any partial object at the beginning of the page aleviates this. The encoding fits within a single 32 bit word and may be naturally incorporated into the stable stores mapping tables.

8.3.4. Pointer Quarantine for Distribution and Garbage Collection

Use of page protection mechanisms allows the language level support to manage the quarantine of pointer references without impacting upon the structure of the user code itself. The is quarantine owes much to the mechanisms used to implement generation garbage collection. The page-cards mechanism [Hosking and Hudson 1993] allows native code to run within a system supporting generational garbage collection without requiring the user code to explicitly check for creation of inter-generational references.

Performance of programs which access a distributed shared persistent address space is improved if each program uses a local heap area in which to perform the majority of its computations. These areas may be separately garbage collected in a similar manner to generation garbage collection schemes so long as inter-area pointers are strictly prohibited. An integration of the action of the distributed coherency algorithm with a variant of the page-cards mechanism allows these pointers to be controlled.

8.3.5. Persistence Orthogonal to native code

The particular strengths of the page based approach come about through the transparency to user level code. User code can be written which is oblivious to the action of the store, pointer quarantine, and distribution mechanisms.

This transparency is best illustrated in the production of native code. Considerable performance gains are possible over a pure software implementation if, rather than produce code for execution by a high level interpreter, code directly executable by the underlying machine architecture is generated. Two approaches are possible when generating such native code. One is to produce code which includes in-line code to manufacture the illusion of orthogonal persistence. Such code includes explicit tests for object residency, tests to ensure that store resiliency is not compromised, and code to maintain the structures used by the garbage collection system. Alternatively, by utilising the techniques outlined in this thesis, native code can be produced which contains no such tests. The illusion of persistence being transparently effected in response to page faults, and maintenance of

garbage collection structures in response to protection exceptions. The produced code is smaller and is required to perform less computation to effect the same result. Arguably these benefits are simply an example of reaping the rewards of orthogonal persistence.

These are strong claims, however the ultimate balance of performance between a pure software implementation and one which is based upon page based mechanisms lies in the answer to a simple question.

Which is the better approach?

- To perform a large number of simple tests and occasionally explicitly call support functions that are optimised for the particular task.

or

- To perform no tests but to occasionally incur the cost of an exception and to call a support function which due to the poorer fit of the page based model to the language system may not be so optimal for the task.

The solution to this question is not clear and must wait for the production of suitable bench-marking suites [Atkinson, Dirnie et al. 1992; Carey, DeWitt et al. 1993]. The answer must be dependant upon many individual features of the host architectures and target languages.

8.4. Store Design

Persistent stores must provide a resilient, stable repository for the entire persistent state of a system. Techniques for maintaining these goals include snapshots of the entire system state at once, logging changes made to the store, and logging high level input to the user program. Each is capable of recovering a consistent view of the persistent store in the face of system failure. Combinations of these mechanisms allow multiple stores to be linked to form large persistent systems. In particular, logging systems can provide valuable flexibility to systems attempting to co-ordinate the actions of separate stores into a consistent whole.

Page based store design has mostly been oriented around the shadow store mechanisms of Lorie, although log based approaches are also used. The Casper system described provides three different shadow store designs.

- The basic store provides a persistent virtual address space in which Napier88 programs are supported. Its design allows complied native code Napier88 programs to execute in a manner oblivious to the action of the store.

- An enhanced design capable of allowing separate parts of the persistent address space to meld independently of the remainder of the space. This requires special care of the stores internal structures to avoid corrupting the store.
- Another variant allows an arbitrary number of versions of the persistent space to be held. This allows the store to be integrated into higher level global consistency mechanisms.

Efficiency of implementations is hampered by the inappropriate mechanisms provided by host operating systems. In particular, utilising the Unix operating system's ability to map files into the virtual address space although initially appealing choice has significant costs.

- A design limitation forces the store manager to copy each page into and then onto a temporary buffer when that page is modified.
- An unnecessary page fault, and consequent read from disk is incurred.
- No control over the layout of the store on disk is available and thus only limited action can be taken to improve store performance by clustering disks writes is available.

Implementations based upon the Mach operating system's external pager are able to avoid many of these problems, although the external pager is not ideal either. The external pager incurs the cost of extra copying of data as it is transferred to the kernel, and only provides limited control of eviction of data from physical memory.

Stores that utilise memory mapped files or external pager designs eliminate unnecessary swapping of data between physical memory and the operating systems swap space. Designs which utilise object based movement, or designs which modify the data before presenting it in addressable memory to the user program are unable to avoid these costs. To enable such integration without compromising the integrity of the persistent store shadow paging mechanisms are used to ensure that a self consistent version of the persistent state is alway available on stable media.

8.5. Distribution

Distribution of persistent systems is attacked on two separate fronts in this thesis. The first in describing tactics for providing distributed access to a Napier88 system can be provided using page based DSM techniques. The second front is that of providing an environment in

which concurrent and distributed systems can be implemented and coordinated to form a consistent whole.

8.5.1. Distributed Casper

The Distributed Casper system allows individual client programs, perhaps on distributed machines, to access and share a single persistent address space. Many of the mechanisms described in the earlier chapters are utilised in Casper to make the provision of this space tractable. In particular:

- Individual clients maintain a local heap which may be separately garbage collected and to which they are guaranteed exclusive access. The distributed shared memory coherency algorithms interact with the garbage collection and stability mechanisms to ensure that the integrity of these local heaps is maintained.
- The system tracks interdependencies generated through shared access to data and allows subsets of the clients that are independent of the remaining clients to meld together. This requires use of the previously described special Casper store.

The Casper system utilises the page based coherency system to implement a form of atomic action without using potentially expensive mutex locking across a network. By selectively denying access to other clients of the shared space when the system is required to perform a logically atomic operation, unnesessary communication with other clients can be avoided.

Casper tracks interactions between individual clients of the persistent address space. To improve performance of the store, only those clients that are interdependent by virtue of having accessed modified data are forced to meld together. Independent clients continue to execute independently.

8.5.2. Grasshopper

Grasshopper is intended to provide a new operating system in which persistence is provided as an intrinsic attribute. In particular the Grasshopper design is intended to provide an environment in which the page based techniques for the provision of language level and store abstractions are natural.

Application programs only perceive a resilient persistent address space. Any programming language or application system may reap the benefits of orthogonal persistence without change. The persistence and resilience of individual address spaces is

the responsibility of user level entities called managers. These work in conjunction with the Grasshopper kernel to provide a global resilient system state which may be restored when a failure occurs. This architecture provides a framework in which individual managers may, if they wish, provide persistence in different ways without the need to explicitly manage global consistency.

Managers may use logging, both of store changes and high level input as well as shadow paging techniques. The kernel based recovery mechanisms are able to exploit the extra freedom available from such techniques to create consistent recoverable system states. Log based recovery mechanisms require that execution be deterministic. Rather than place special requirements upon the kernel, the use of a proxy mechanisms allows managers to control execution within containers, allowing the managers to enforce deterministic execution and to log external interactions with no impact on the structure of the code within containers.

8.6. Conclusions.

The abstraction of orthogonal persistence is a powerful tool in building programming systems. Whilst attempts have been made to utilise specialised hardware and to build software based systems above existing operating systems this thesis has described a spectrum of techniques by which the peculiar attributes of demand paged virtual memory can be exploited in the provision of orthogonal persistence.

These techniques allow language level mechanisms to be transparently supported, allow for fast and efficient movement of data to and from resilient stable stores and provide mechanisms by which distributed access to data by be provided. A logical cumulation of these efforts is the delivery of an operating system whose design is tailored to support these techniques and in which the provision of an orthogonally persistent programming environment is natural.

8.6.1. Status

The Casper project has concluded and has successfully demonstrated the utility of page based techniques in the provision of distributed access to Napier88 programming environments. Its legacy lives on in the Grasshopper system where it will be used to provide a ready made Napier88 system.

The Grasshopper project is currently under intensive development on Alpha AXP systems. At the time of writing, the kernel is itself persistent, contains device drivers, a

scheduler supporting multiple loci and support for containers. Operation of the first managers (based upon the Casper multi-phase design) is entering debugging. Great things are expected.

Appendix A.

This appendix provides explanation of the various states shown in the Client FSA (Figure 53).

STATE	EXPLANATION
START not held	The START state represents the entry point for any page not held by the client.
Wait for Page Supply	A copy of the page has been requested by the client; a supply page signal is expected to arrive containing an up-to-date copy of this page.
(XRM state) Wait for Page Supply	XRM indicates that an external read request followed by an external modify request may be necessary for the client to gain the required access to the page. This is a result of the interpreter attempting to modify a non-resident page. A copy of the page has already been requested via an XR signal; upon receipt of this page, modification permission must be sought. A supply page signal is expected to arrive which will contain an up-to-date copy of the page.
SH MOD	The page is shared with other clients and is modified with respect to the stable copy in the store.
SH NONMOD	The page is shared with other clients and has not been modified since it was last stabilised.
NONSH NONMOD	No other client holds a copy of this page. The page has not been modified since it was last stabilised.
NONSH MOD	This client is the only client with a copy of the page; the page has been modified with respect to the stable copy.
LOCAL	The page is a member of the client's local heap. It is implicitly non-shared and open for read and write access.
SH Waiting for WRACK	The interpreter has attempted to modify a shared page; thus, write permission has been requested. A write acknowledgement signal is expected in reply to this request.

This table provides explanation of the various signals used in the Client FSA (Figure 53).

SIGNAL	EXPLANATION
Incoming to the Client Request Handler	
→CR	An internal client read request from the client's own interpreter.
→CM	An internal client modify request from the client's own interpreter.
→XR	An external read request from another client via the Stable Store Server.
→XM	An external modify invalidation request from the Stable Store Server, as a result of another client's request to modify.
→WRACK	A write acknowledgement signal from the Stable Store Server to indicate modification permission is being granted.
→SP	A supply page signal from the Stable Store Server or a client, in reply to an external read request.
→STAB	A stabilisation request from the Stable Store Server.
Outgoing from the Client Request Handler	
XR→	An external read request from this client to the Stable Store Server.
XM→	An external modify request from this client to the Stable Store Server.
MODXM→	An external modify request for a previously modified page from this client to the Stable Store Server.
INVACK→	An invalidation acknowledgement from this client to the Stable Store Server, in reply to an external modify invalidation request.
MOD→	A modification signal from this client to the Stable Store Server to inform the Stable Store Server of this client's intention to modify the page.
SP→	A supply page signal, including an up-to-date copy of the page, from this client to another client in reply to an external read request.

This table provides explanation of the various states shown in the Server FSA (Figure 61).

STATE	EXPLANATION
START No entry	The START state represents the entry point for any persistent page which has not been exported from the store.
Non existent	A page moves from this state when it is allocated for local heap or copy-out usage; a page belonging to this state would not hold any persistent objects.
NONSH NONMOD	A copy of the page is held by one client only and it has not been modified with respect to the stable copy.
NONSH MOD	One client holds a copy of the page and this copy is a modified version of the stable copy.
NONSH MOD copy avail	Only one client holds a copy of the page, which has been modified with respect to the stable copy. An up-to-date copy of this page has been sent back to the stable store.
SH NONMOD	At least two clients hold copies of this page. The page has not been modified with respect to the stable copy.
SH MOD	Two or more clients hold up-to-date copies of this page, which has been modified with respect to the stable copy.
SH MOD copy avail	This page is shared by two or more clients and has been modified since the last stabilisation in which it was involved. An up-to-date copy of this page has been sent to the stable store and is available for supply to further requesting clients.
SH NONMOD page avail	A copy of the page is held by more than one client.; this copy is non-modified with respect to the stable copy. The Stable Store Server may safely supply further copies of this page directly to requesting clients as the holding clients will not attempt to modify the page without informing the Stable Store Server of their intention to do so.
Wait for INVACK $XRq = \emptyset$	A client has requested modification permission for a shared page. The Stable Store Server has informed other holding clients to invalidate their copies of this page. The Stable Store Server is now awaiting those clients' invalidation acknowledgements. No further read requests have been received for the page.
Wait for INVACK $\lvert XRq \rvert > 0$	The Stable Store Server is awaiting invalidation acknowledgements from those clients which have been requested to invalidate their copies of this page. Read requests for this page have since been received from one or more other clients. Those requests have been delayed on the local XR queue. This queue must be traversed after write acknowledgement has been forwarded to the client which originally requested modification permission.

This table provides explanation of the various signals used in the Server FSA (Figure 61).

SIGNAL	EXPLANATION
Incoming to the Stable Store Server	
→XR	An external read request from a connected client.
→XM	An external modification request from a holding client.
→MODXM	An external modification request from a holding client which also indicates that the page has been modified previously.
→INVACK	An invalidation acknowledgement from a client which previously held the page, in reply to an invalidating external modification request.
→MOD	This is a modification notification from a client which holds or held the page in a non-shared state to inform the Stable Store Server that page has been modified.
→STAB	A stabilisation request to move the page into a non-modified state as a result of a stabilisation.
→STAB Reply	This is a stabilisation reply from a holding client involved in a stabilisation cycle; this reply includes the return of an up-to-date copy of the page.
→PG REQUEST	This is a page request from a client; it implies that the client requires more free space for local heap or copy-out usage.
Outgoing from the Stable Store Server	
XR→	Forwarding of an external read request to a client which holds an up-to-date copy of the page.
SP→	Send a supply page signal to a requesting client, in response to an incoming external read request.
XM→	This corresponds to the forwarding of an invalidating external modification request to all clients holding the page, except the one which requested modification permission.
WRACK→	Here, a write acknowledgement is being forwarded to the client which originally requested modification permission.
NEW PAGE→	The new page signal is sent to supply a new, empty page range to a client, in response to the client's page request.
Miscellaneous	
XRqueue[+Cl]	This represents addition of a client to the local XR queue, since modification permission is being sought for another client.

References

Abrossimov, V., Rozier, M, Shapiro, M. (1989). "Generic Virtual memory management for Operating System kernels." *Operating Systems Review - SIGOS* **23**(5):123-135.

Acceta, M., Baron, R., Bolosky, W., Golub D., Rashid, R., Tevanian, A., Young, M. (1986). "Mach: A New Kernel Foundation for Unix Development." *Proceedings, Summer Usenix Conference* , pages 93-112.

Agrawal, R., Gehani, N. (1989) ODE (Object Database and Environment): The Language and the Data Model. *Sigmod Record.* **18**(3):36-45.

Appel, A. (1989). "Simple Generational Garbage Collection and Fast Allocation." *Software - Practice and Experience* **19**:171-183.

Appel, A. and Li K.(1991). *Virtual Memory Primitives for User Programs*. Architectual Support for Programming Languages and Systems -IV, ACM, pages 96-107.

Archibald, J. and Baer, J. L.(1986). "Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model." *ACM Transactions on Computer Systems* (4):273-298.

Atkinson, M. P. (1978). *Programming Languages and Databases*. Fourth IEEE International Conference on Very Large Databases, IEEE, pages 408-419.

Atkinson, M., Bailey, P., Chisholm, K., Cockshott, W., Morrison, R. (1983). "An Approach to Persistent Programming." *The Computer Journal* **26**(4):360-365.

Atkinson, M., Bailey, R., Chisholm, K., Cockshott, W., Morrison, R. (1983). The Persistent Object Management System. Universities of Glasgow and St Andrews.

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

Atkinson, M., Chisholm, K., Cockshott, W. (1981). "PS-algol: An Algol with a Persistent Heap." *ACM SIGPLAN Notices* **17**(7):24-31.

Atkinson, M., Chisholm, K., Cockshott, W. (1983). "CMS - A Chunk Management System." *Software Practice and Experience* **13**(3):259-272.

Atkinson, M., Dirnie, A., Jackson, N., Philbrow, P. (1992). *Measuring Persistent Object Systems*. 5th International Workshop on Persistent Object Systems, A. Albano and R. Morrison editors. Springer-Verlag, pages 63-85.

Ballard, S. and S. Shrirron (1983). The Design and Implementation of VAX/Smalltalk-80. *Smalltalk-80 Bits of History, Words of Advice.* G. Kranser editor, Addison Wesley, pages 127-150.

Bancilhon, F. and D. Maier (1989). Multilanguage Object-oriented systems: New answers to old database problems. *Future Generation Computers II.* North Holland.

Beloff, B., McIntyre, D., Drummond, B. (1988). Rekursiv Hardware. Linn Smart Computing Ltd.

Bernstein, P., Hadzilacos, V., Goodman, N. (1987). *Concurrency Control and Recovery in Database Systems*. Addison Wesley.

Berry, D. (1971). *Block Structure: Retention or Deletion*? Third Annual ACM Symposium on the Theory of Computing, pages 86-100.

Bertis, V., Truxal, C., Ranweiler, J. (1978). "System/38 Addressing and Authorisation." *I.B.M. System/38 Technical Developments* , pages 51-54.

Birrell, A. D. and B. J. Nelson (1984). "Implementing Remote Procedure Calls." *ACM Transactions on Computer Systems* **2**(1):39-59.

Bishop, P. (1977). Computer Systems with a Very Large Address Space and Garbage Collection. Phd Thesis. Massachusetts Institute of Technology. MIT/LCS/TR-178.

Brown, A. L. (1988). Persistent Object Stores. PhD Thesis. Universiy of St. Andrews. PPRR-71.

Brown, A. L. (1994). "Dynanically Configurable Persistent Object Caches." *Australian Computer Science Communications* **16**(1):629-638.

Brown, A., Carrick, R., Conner, R., Dearle, A., Morrison, R. (1988). The Persistent Abstract Machine. Universities of Glasgow and St Andrews. PPRR-59-88.

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

Brown, A. L. and W. P. Cockshott (1985). CPOMS – A Revised Version of the Persistent Object Management System in C. Universities of Glasgow and St Andrews. PPRR-13-85.

Brown, A. L. and J. Rosenberg (1990). *Persistent Programming Systems: An Implementation Technique*. Proceedings of the 4th International Workshop on Persistent Object Systems, A. Dearle, G. Shaw and S. Zdonik editors. Morgan-Kaufmann, pages 199-212.

Bushell, S., Dearle, A., Brown, A., Vaughan, F. (1994). *Using C as a Compiler Target Language for Native Code Generation in Persistent Systems.* 6th International Workshop on Persistent Object Systems, M. Atkinson, D. Maier, V. Benzaken editors. Springer Verlag, pages 164-183.

Campbell, R., Johnston, G., Russo, V. (1987). "Choices (Class Hierarchical Open Interface for Custom Embedded Systems." *ACM Operating Systems Review* **21**(3): 9-17.

Cardelli, L. (1985). Amber. AT&T Bell Labs. Technical Report Car85

Carey, M., DeWitt, D., McNaughton, J. (1993). *The 007 Benchmark.*. Proceeding of the ACM SIGMOD Conference on Management of Data. pages 12-21.

Challis, M. F. (1978). Database Consistency and Integrity in a Multi-User Environment. *Databases: Improving Useability and Responsiveness.* Academic Press. pages 245-270.

Chandramohan, T. and H. Levy (1994). "Hardware and Software Support for Efficient Exception Handling." University of Washington, Seattle, Technical Report 94-07-05

Chen, P., Lee, E., Gibson, G., Katz, R., Patterson, D. (1994). "RAID: High-Performance, Reliable Secondary Storage." *Computing Surveys* **26**(2):145-185.

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

Cockshott, W. P. and P. W. Foulk (1990). *Implementing 128 Bit Persistent Addresses on 80x86 Processors.* Proceedings of the International Workshop on Computer Architectures to Support Security and Persistence of Information, Springer-Verlag and British Computer Society. pages 123-136.

Cook, J., Wolf, A., Zorn, B. (1994). Partition Selection Policies in Object Database Garbage Collection. *SIGMOD Record.* **23**(2):370-382.

Dasgupta, P., LeBlanc, R., Mustaque, A., Umakishore, R. (1988). The Clouds Distributed Operating System. Arizona State University. Technical Report 88-25.

Dearle, A. (1988). On the Construction of Persistent Programming Environments. University of St. Andrews. PPRR-65.

Dearle, A., Rosenberg, J., Henskens, F., Vaughan, F. (1992). *An Examination of Operating System Support for Persistent Object Systems.* 25th Hawaii International Conference on System Sciences, Poipu Beach, Kauaii, IEEE Computer Society Press, pages779-789.

Dearle, A., Rosenberg, J., Vaughan, F. (1991). *A Remote Execution Mechanism for Distributed Homogeneous Stable Stores*. Third International Workshop on Database Programming Languages, Morgan Kauffman, pages 125-138.

Draves, R., Jones, M., Thompson, M. (1988). MIG - The Mach Interface Generator. Dept of Computer Science, Carnegie-Mellon University. Technical Report DJT88.

Ellis, J., Li, K., Appel, A. (1988). Real-time Concurrent Collection on Stock Multiprocessors. DEC SRC. Report #25.

Eppinger, J., Mummert, L., Spector, A. (1991). *Camelot and Avalon*. San Mateo, California, Morgan Kaufmann.

Eric Koldinger, H. L., Jeffrey Chase and Susan Eggers (1991). The Protection Lookaside Buffer: Efficient Protection for Single Address-Space Computers. University of Washington. Technical Report 91-11-05.

Eswaran, K., Gray, J. Lorie, R., Traiger, I. (1976). "The Notions of Consistency and Predicate Locks in a Database System." *Communications of the Association for Computing Machinery* **19**(11):624 - 633.

Fabry, R. (1974). "Capability-Based Addressing." *Communications of the ACM* **17**(7): 403-412.

Fabry, R. S. (1973). *The Case for Capability Based Computers*. Fourth Symposium on Operating Systems Principles, ACM.

Farkas, A. (1994). Program Construction and Evolution in a Persistent Integrated Programming Environment. PhD Thesis. University of Adelaide.

Fidge, C. (1988). *Timestamps in Message-Passing Systems That Preserve Partial Ordering*. 11th Australian Computer Science Conference pages 56-66.

Fotheringham, J. (1961). "Dynamic Storage Allocation in the Atlas Computer Including an Automatic Use of a Backing Store." *CACM* **4**(10):435-436.

Goldberg, A. and D. Robson (1983). *Smalltalk-80: The language and its Implementation*. Addison Wesley.

Gray, J. (1990). A Cencus of Tandem System Availability Between 1985 and 1990. Tamdem Computers. Technical Report 90.01.

Harland, D. M. (1988). *REKURSIV: Object-oriented Computer Architecture*. Ellis-Horwood Limited.

Harty, K. and D. R. Cheriton (1992). *Application-Controlled Physical Memory Using External Page-Cache Management*. Architectural Support for Programming Languages and Operating Systems-V, ACM.

Hawking, S. W. (1988). *A Brief History of Time: From the Big Bang to Black Holes*. New York, Bantam.

Henessy, J. L. and D. A. Patterson (1990). *Computer Architecture: A Quantitative Approach*. San Mateo, Morgan Kaufmann.

Henskens, F. A. (1992). *Addressing moved modules in a capability based distributed shared memory*. 25th Hawaii International Conference on System Sciences, IEEE, pages 769-778.

Henskens, F., Rosenberg, J., Keedy, J. (1991). *A Capability-based Distributed Shared Memory*. Proceedings of the 14th Australian Computer Science Conference pp29.1-29.12.

Hosking, A. (1991). "Main Memory Management for Persistence." Position Paper, OOPSLA '91, Worshop on Garbage Collection.

Hosking, A., Brown, E., Moss, J. (1993). *Update Logging for Persistent Programming Languages: A Comparative Performance Evaluation*. 19th International Conference on Very Large Data Bases, Morgan Kaufmann, pages 429-440.

Hosking, A. and Hudson, R. (1993). *Remembered sets can also play cards.* OOPSLA '93, Workshop on Memory Management and Garbage Collection, ACM.

Hosking, A. and Moss, J. (1993). *Object Fault Handling for Persistent Programming Languages: A Performance Evaluation.* Proceedings ACM Conference on Object-Oriented Programming Systems, Languages and Applications, ACM. pages 288-303.

Hosking, A., and Moss, J. (1993). *Protection traps and alternatives for memory management of an object-oriented language.* Proceedings 14th SOSP, Asheville NC, pages 1-14.

Apple Computer Inc. (1986). "Inside Macintosh." Addison Wesley

Ingalls, D. H. H. (1983). The Evolution of the Smalltalk Virtual Machine. *Smalltalk-80 Bits of History, Words of Advice.* G. Kranser editor, Addison Wesley. pages 9-28.

Irlam, G. (1992). Private Communication.

Jallili, R. and F. Henskens (1995). *Entity Dependency in Stable Distributed Persistent Stores.* Distribution and Concurrency in Persistent Systems Minitrack. Hawaii International Conference on Systems Science.

Jeffrey Chase, H. L., Michael Feeley and Edward Lazowska (1993). Sharing and Protection in a Single Address Space Operating System. *ACM Transactions on Computer Systems* **12**(2)

Johnson, D. and W. Zwaenepoel (1990). "Recovery in Distributed Systems Using Optimistic Message Logging and Checkpointing." *Journal of Algorithms* **11**(3):462-491.

Kaehler, T. and G. Krasner (1983). LOOM – large object-oriented memory for Smalltalk-80. *Smalltalk-80: Bits of History, Words of Advice.* G. Kranser editor, Addison-Wesley. pages 251-270.

Kalsow, B. and E. Muller (1991). SRC Modula-3. DEC Systems Research Centre.

Kane, G. and J. Heinrich (1992). *MIPS RISC Architecture*. Prentice-Hall.

Kernighan, B. W. and D. M. Ritchie (1978). *The C programming language*. Prentice-Hall.

Khalidi, Y. and Nelson, M. (1993). The Spring Virtual Memory System. Sun Microsystems Laboratories.

Kirby, G., Connor, R., Cutts, Q., Dearle, A., Farkas, A., Morrison, R. (1992). *Persistent Hyper-Programs*. 5th International Workshop on Persistent Object Systems, Persistent Object Systems, A. Albano and R. Morrison editors. Springer-Verlag, pages 86-106.

Koch, B., T. Schunke, Dearle, A., Vaughan, F., Marlin, C., Fazakerley, R., Barter, C. (1990). *Cache Coherence and Storage Management in a Persistent Object System.* Proceedings, The Fourth International Workshop on Persistent Object Systems, A. Dearle, G. Shaw and S. Zdonik editors. Morgan Kaufmann, pages 99-109.

Kolodner, E., Liskov, B., Weihl, W. (1989). *Atomic Garbage Collection: Managing a Stable Heap*. Proceeding of the 1989 ACM SIGMOD Conference on the Managment of Data, pages 15-25.

Kranser, G. editor (1983). *Smalltalk-80 Bits of History, Words of Advice*. Addison Wesley.

Lamb, C., Landis, G., Orenstein, J., Weinreb, D. (1991). "The Objectstore Database System." *CACM* **34**(10):50-63.

Lamport, L. (1978). "Time, Clocks, and the Ordering of Events in a Distributed System." *CACM* **21**(7): 558-565.

Leffer, S., McKusick, M., Karels, M., Quarterman, J. (1989). *The Design and Implementation of the 4.3BSD Unix Operating System.* Reading, Massachusetts, Addison Wesley.

Levy, H. and P. Lipman (1982). "Virtual Memory Management in the VAX/VMS Operating System." *Computer* **15**(3): 35-41.

Li, K. and P. Hudak (1989). "Memory Coherence in Shared Virtual Memory Systems." *ACM Transactions on Computing Systems* **7**(4): 321-359.

Lieberman, K. and Hewitt, C. A real Time Garbage Collector Based Upon the Lifetime of Objects. *Communications of the ACM*, **26**(6):419-429

Lin, S. (1970). *An Introduction to Error-Correcting Codes*. Englewood Cliffs, N.J., Prentice-Hall.

Lindström, A., Dearle, A., diBona, R., Farrow, M., Henskens, F., Rosenberg, J., Vaughan, F. (1994). *A Model For User-Level Memory Management in a Distributed, Persistent Environment.* 17th Australian Computer Science Conference, pages 343-354.

Lindström, A., Dearle, A., di Bona, R., Rosenberg, J., Vaughan, F. (1994). User-level Management of Persistent Data in the Grasshopper Operating System. University of Sydney and University of Adelaide. Technical Report GH-08.

Lindström, A., di Bona, R., Dearle, A., Norris, S., Rosenberg, J., Vaughan, F. (1994). *Persistence in the Grasshopper Kernel*. 6th International Conference on Persistent Object systems, M. Atkinson, D. Maier, V. Benaken editors. Springer Verlag. pages 60-78.

Liskov, B., Curtis, D., Johnson, P., Scheifler, R. (1987). "Implementation of Argus." *ACM Operating Systems Review* **21**(5): 111-122.

Liskov, B. H. and S. N. Zilles (1974). "Programming with Abstract Data Types." *SIGPLAN Notices* **9**(4): 50-59.

Lo Basso, T., Vaughan, F., Dearle, A., Marlin, C., Barter, C. (1991). A Persistent Object System Shared by Multiple Concurrent Clients. University of Adelaide. Technical Report PS-01.

Lorie, R. A. (1977). "Physical Integrity in a Large Segmented Database." *Association for Computing Machinery Transactions on Database Systems* **2**(1): 91-104.

Malhorta, A. and S. Munroe (1992). *Support for Persistent Objects: Two Architectures.* 25th Annual Hawaii International Conference on System Sciences, Hawaii, University of Hawaii.

Mattern, F. (1989). Virtual Time and Global States in Distributed Systems. *Parallel and Distributed Algorithms.* Elseweir/North Holland. pages 215-226.

Mattern, F. (1990). Efficient Distributed Snapshots and Global Virtual Time Algorithms for Non-FIFO Systems. University of Kaiserslauten Dept of Computer Science.

Morrison, R. (1982). "S-algol: a simple algol." *BCS Computer Bulletin* **2**(31): 17-20.

Morrison, R., Brown, A., Carrick, R., Conner, R., Dearle, A., Livesey, M., Barter, C., Hurst, A. (1989). *Language Design Issues in Supporting Process-Oriented Computation in Persistent Environments*. Proceedings of the Twenty-Second Annual Hawaii International Conference on System Sciences, pages 736-744.

Morrison, R., Brown, A., Conner, R., Cutts, Q., Kirby, G., Dearle, A., Rosenberg, D., Stemple, D. (1990). Protection in Persistent Object Systems. *Security and Persistence.* Springer-Verlag. pages 48-66.

Morrison, R., Brown, A., Conner, R., Dearle, A. (1989). The Napier88 Reference Manual. University of St Andrews. PPRR-77-89.

Moss, E. and A. Sinofsky (1988). Managing Persistent Data with Mneme: Designing a Reliable, Shared Object Interface. *Advances in Object-Oriented Database Systems.* berlin, Springer-Verlag. pages 298-316.

Moss, J. E. B. (1989). *Addressing Large Distributed Collections of Persistent Objects: The Mneme Project's Approach.* 2nd International Workshop on Database Programming Languages, San Mateo, California, R. Hull, D. Stemple editors. Morgan Kaufmann. pages 358-374.

Moss, J. E. B. (1990). "Design of the Mneme System." *Transactions on Information Systems* **8**(2): 103-139.

Moss, J. E. B. (1991). "Working with Persistent Objects: To Swizzle or Not to Swizzle." University of Massachesetts, Technical Report 90-38.

Munro, D. S. (1993). On the Integration of Concurrency, Distribution and Persistence. PhD Thesis. University of St Andrews. Research Report CS/94/1.

Object Design (1994). ObjectStore Technical Overview. Object Design Inc.

Pountain, D. (1988) Rekursiv: An Object-Oriented CPU. *Byte*, November 1988, pages 341-349.

PS-algol (1985). PS-algol Abstract Machine Manual. Universities of Glasgow and St Andrews.

PS-algol (1988). PS-algol Reference Manual - fourth edition. University of Glasgow and St Andrews.

Rashid, R., Tevanian, A., Young, M., Golub, D., Baron, R., Black, D., Bolosky, W., Chew, J. (1988). *Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures*. IEEE Transactions on Computers, **37**(8):896-908.

Richardson, J. and M. Carey (1989). "Persistence in the E Language: Issues and Implementation." *SPE* **19**(12): 1115 - 1150.

Ritchie, D. M. and K. Thompson (1978). "The UNIX Time-Sharing System." *The Bell System Technical Journal* **63**(6): 1905-1930.

Rosenberg, J. (1991). *Architectural Support for Persistent Object Systems*. International Workshop on Object-Orientation in Operating Systems, Xerox-Parc, California, IEEE Computer Society Press.

Rosenberg, J. and D. A. Abramson (1985). *MONADS-PC: A Capability Based Workstation to Support Software Engineering*. 18th Hawaii International Conference on System Sciences, pages 515-522.

Rosenberg, J., Henskens, F., Brown, A., Morrison, R., Munro, D. (1990). *Stability in a Persistent Store Based on a Large Virtual Memory.* Proceedings of the International Workshop on Architectural Support for Security and Persistence of Information, J. Rosenberg, J.L. Keedy editors. Springer-Verlag and British Computer Society. pages 229-245.

Rosenberg, J. and J. L. Keedy (1987). *Object Management and Addressing in the MONADS Architecture*. Proceedings of the International Workshop on Persistent Object Systems.

Rosenberg, J., Keedy, J., Abramson, D. (1992). "Addressing Mechanisms for Large Virtual Memories." *The Computer Journal* **35**(4): 369-375.

Ross, D. M. (1983). Virtual Files: A Framework for Experimental Design. University of Edinburgh. Technical Report CST-26-83.

Satyanarayanan, M., Mashburn, H., Puneet, K., Steere, D., Kistler, J. (1994). "Lightweight Recoverable Virtual Memory." *Transactions on Computer Systems* **12**(1): 33-57.

Schwarz, R. and F. Mattern (1992). Detecting Causal Relationships in Distributed Computations: In Search of The Holy Grail. Department of Computer Science, University of Kaiserslautern and University of Saarland. Technical Report SFB-124/15/92.

Singhal, V., Sheetal, V., Wilson, P. (1992). *Texas: An Efficient, Portable Persistent Store*. 5th International Workshop on Persistent Object Systems, A. Albano, R.Morrison editors. Springer-Verlag, pages 11-33.

Sistla, A. and J. Welch (1989). "Efficient Distributed Recovery Using Message Logging." 8th Annual ACM Symposium on Principles of Distributed Computing. pages 223-238.

Sites, R. (1993). "Alpha AXP Architecture." *Communications of the ACM* **36**(2): 33-83.

Stamos, J. and D. Gifford (1990). "Remote Evaluation." *TOPLAS* **12**(4): 537-565.

Stemple, D. and R. Morrison (1992). *Specifying Flexible Concurrency Control Schemes: An Abstract Operational Approach.* Proceedings of the 15th Australian Computer Science Conference. pages 873-891.

Strom, R. and S. Yemini (1985). "Optimistic Recovery in Distributed Systems." *ACM Transactions on Computer Systems* **3**(3): 204-226.

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

Symbolics, Documentation Group. (1984). *Symbolics 3600 Technical Summary.* Symbolics Inc.

T Kilburn, D. E., MJ Lanigan and FH Sumner (1962). "One-Level Storage System." *IRE Trans* Vol **2**, pages 223-235.

Tam, M., Smith, J., Farber, D. (1990). "A Taxonomy-based Comparison of Several Distributed Shared Memory Systems." *Operating Systems Review* **24**(3): 40-67.

Tanenbaum, A. S. (1987). *Operating Systems: Design and Implementation*. Prentice Hall.

Thatte, S. M. (1986). *Persistent Memory: A Storage Architecture for Object Oriented Database Systems*. Proceedings of the ACM/IEEE International Workshop on Object-Oriented Database Systems, pages 148-159.

Ungar, D. and D. Patterson (1983). Berkeley Smalltalk: Who Knows Where the Time Goes? *Smalltalk-80 Bits of History, Words of Advice.* G. Kranser editor, Addison Wesley. pages 189-206.

Vaughan, F. and A. Dearle (1992). *Supporting Large Persistent Stores Using Conventional Hardware.* 5th International Workshop on Persistent Object Systems, A.Albano, R. Morrison editors. Springer-Verlag, pages 34-53.

Vaughan, F., Dearle, A., Cao, J., di Bona, R., Farrow, M., Henskens, F., Lindström, A., Rosenberg, J. (1994). *Causality Considerations in Distributed Persistent Operating Systems*. 17th Australian Computer Science Conference, Australian Computer Science Communications, Vol 16, pages 409-420.

Vaughan, F., Schunke, T., Koch, B., Dearle, A., Marlin, C., Barter, C. (1990). *A Persistent Distributed Architecture Supported by the Mach Operating System*. Proceedings of the 1st USENIX Conference on the Mach Operating System, pages 123-140.

Vaughan, F., Schunke, T., Koch, B., Dearle, A., Marlin, C., Barter, C. (1992). "Casper: A Cached Architecture Supporting Persistence." *Computing Systems* **5**(3): 337-359.

Welch, B. *The File System Belongs in the Kernel.* Proceedings of the 2nd USENIX Mach Symposium, pages 233-250.

Wilkinson, T., Striemerling, T., Osmon, P., Saulsbury, A., Kelly, P. (1992). *Angel: A Proposed Multiprocessor Operating System Kernel*. European Workshop on Parallel Computing.

Wilson, P. (1990). Pointer Swizzling at Page Fault Time: Efficiently Supporting Huge Address Spaces on Standard Hardware. University of Illinois at Chicago. Techncal Report UIC-EECS-90-6.

Wilson, P. (1991). "Pointer Swizzling at Page Fault Time: Efficiently Supporting Huge Address Spaces on Standard Hardware." *Computer Architecture News* (June): 6-13.

Wu, K.-L. and W. K. Fuchs (1990). "Recoverable Distributed Shared Virtual Memory." *IEEE Transactions on Computers* **39**(4): 460-469.

Wu, K-L., Fuchs, W., Patel, J. (1990) *Error Recovery in Shared Memeory Multiprocessors Using Private Caches.* IEEE Transactions on Parallel and Distributed Systems. **1**(2): 231-240.

Yong, V., Naughton, J., Yu, J. (1994). *Storage Reclamation and Reorganisation in Client-Server Persistent Object Stores.* Proceedings of the 10th International Conference on Data Enginering, pages 120-131.